TELOS Talks – Scheduling & eBPF 101

Zonghao Zhang

2025.6.27

Question-Oriented

This talk is question-oriented.

- What
- Why
- How
- (When, Where, …)

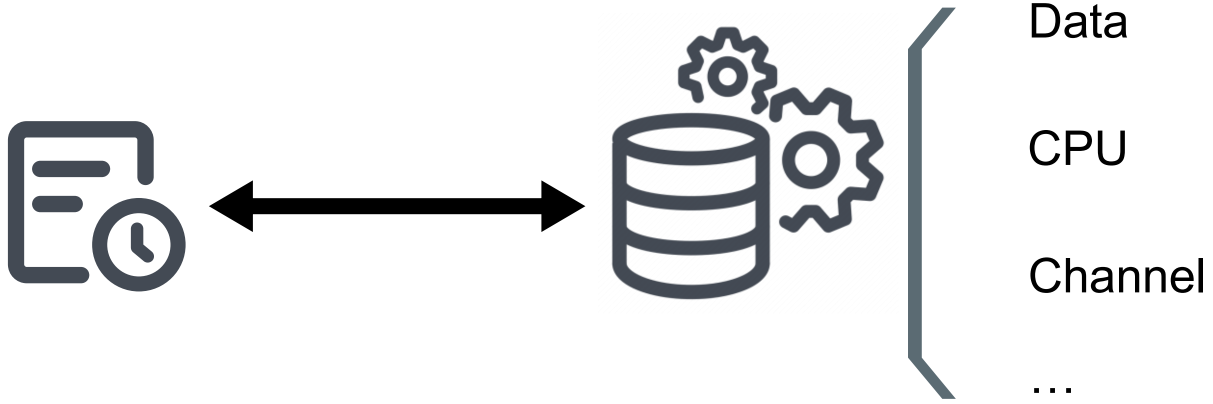
To be free about the questions, as most of them are self Q&A.

Contents

Scheduling - Tasks & Resources

What is the “scheduling”?

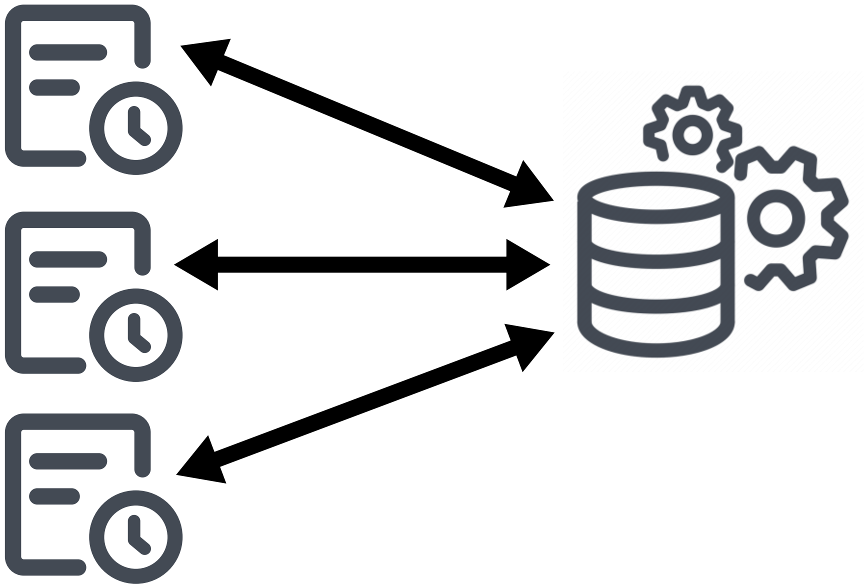
- TL;DR: Mapping work to resources (; and vice versa).



Scheduling - Exclusive & Shared

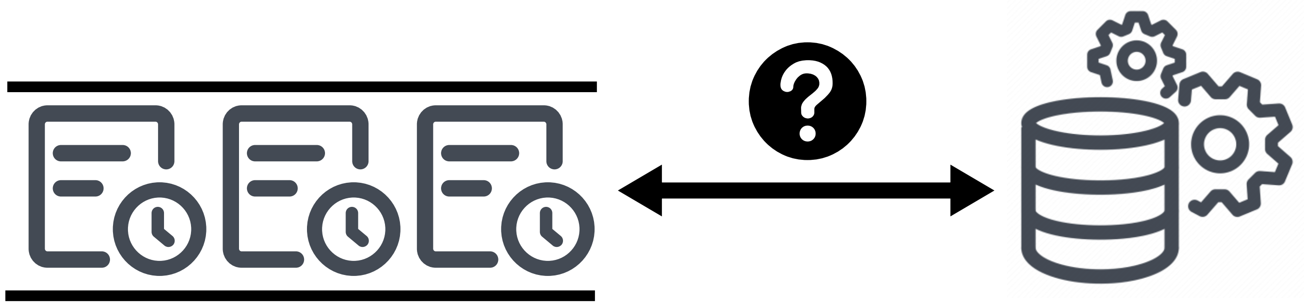
Why we need scheduling?

- Too many mouths to feed.
- Exclusively use shared resource.



Scheduling - Policy & Mechanism

How to make and perform scheduling decisions?



Policy

- How to select the next running task?
- When to make a new decision?

Mechanism

- How to manage pending tasks?
- Where to make decisions?
- How to perform the decision?

Scheduling - Here & There

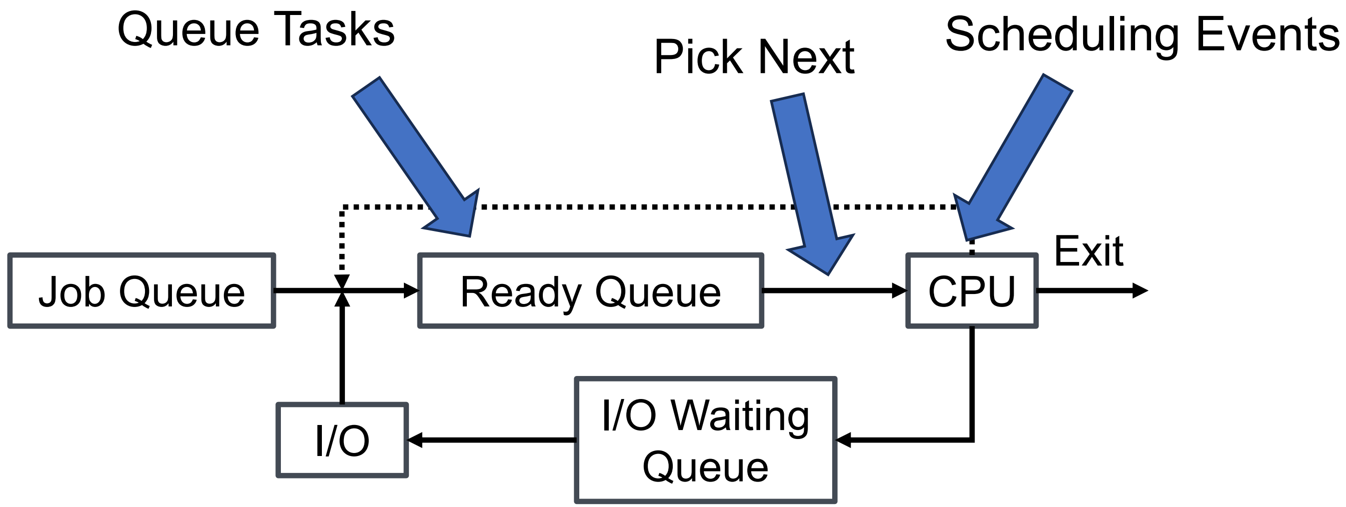
Where is the scheduling required?

- CPU, Network, I/O Request, Memory, Storage, etc.

(This talk primarily focus on CPU and Network)

CPU Scheduling

A typical workflow of CPU scheduling.



Linux Task/Thread Scheduler

Policies

- Default (CFS/EEVDF)
- Real-Time (FIFO, Round-Robin)
- Deadline
- Other (Batch, Idle)

Task Priority

- Static priority (sched_prio)
  - Range from 0 to 99, used in real-time policies
- Dynamic priority (nice)
  - Enabled when sched_prio is zero
  - Range from -20 to 19
  - Lower is more favorable to the scheduler

```c
struct sched_class {

#ifdef CONFIG_UCLAMP_TASK
  int uclamp_enabled;
#endif

  void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
  bool (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
  void (*yield_task)   (struct rq *rq);
  bool (*yield_to_task)(struct rq *rq, struct task_struct *p);

  void (*wakeup_preempt)(struct rq *rq, struct task_struct *p, int flags);

  int (*balance)(struct rq *rq, struct task_struct *prev, struct rq_flags
*rf);
  struct task_struct *(*pick_task)(struct rq *rq);

  struct task_struct *(*pick_next_task)(struct rq *rq, struct task_struct
*prev);

  void (*put_prev_task)(struct rq *rq, struct task_struct *p, struct
task_struct *next);
  void (*set_next_task)(struct rq *rq, struct task_struct *p, bool first);

#ifdef CONFIG_SMP
  int  (*select_task_rq)(struct task_struct *p, int task_cpu, int flags);

  void (*migrate_task_rq)(struct task_struct *p, int new_cpu);

  void (*task_woken)(struct rq *this_rq, struct task_struct *task);

  void (*set_cpus_allowed)(struct task_struct *p, struct affinity_context
*ctx);

  void (*rq_online)(struct rq *rq);
  void (*rq_offline)(struct rq *rq);

  struct rq *(*find_lock_rq)(struct task_struct *p, struct rq *rq);
#endif

  void (*task_tick)(struct rq *rq, struct task_struct *p, int queued);
  void (*task_fork)(struct task_struct *p);
  void (*task_dead)(struct task_struct *p);

  void (*switching_to) (struct rq *this_rq, struct task_struct *task);
  void (*switched_from)(struct rq *this_rq, struct task_struct *task);
  void (*switched_to)  (struct rq *this_rq, struct task_struct *task);
  void (*reweight_task)(struct rq *this_rq, struct task_struct *task,
            const struct load_weight *lw);
  void (*prio_changed) (struct rq *this_rq, struct task_struct *task,
            int oldprio);

  unsigned int (*get_rr_interval)(struct rq *rq,
          struct task_struct *task);

  void (*update_curr)(struct rq *rq);

#ifdef CONFIG_FAIR_GROUP_SCHED
  void (*task_change_group)(struct task_struct *p);
#endif

#ifdef CONFIG_SCHED_CORE
  int (*task_is_throttled)(struct task_struct *p, int cpu);
#endif
};
```

CFS/EEVDF

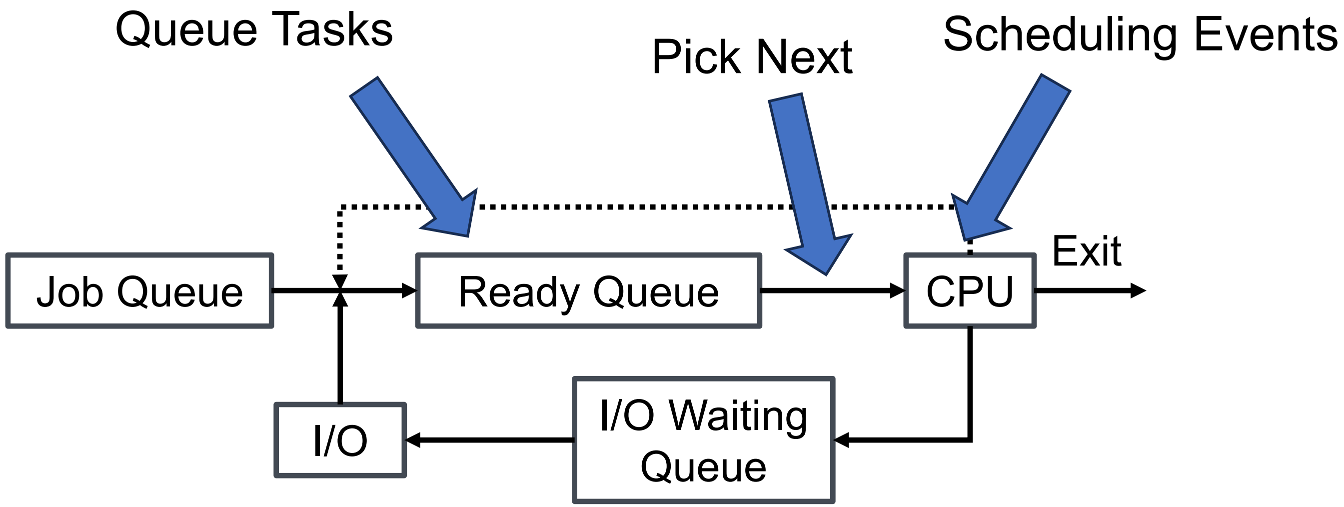
Virtual run time (vruntime)

- Tasks’ real execution time weighted by tasks’ niceness.

CFS/EEVDF

Virtual run time (vruntime)

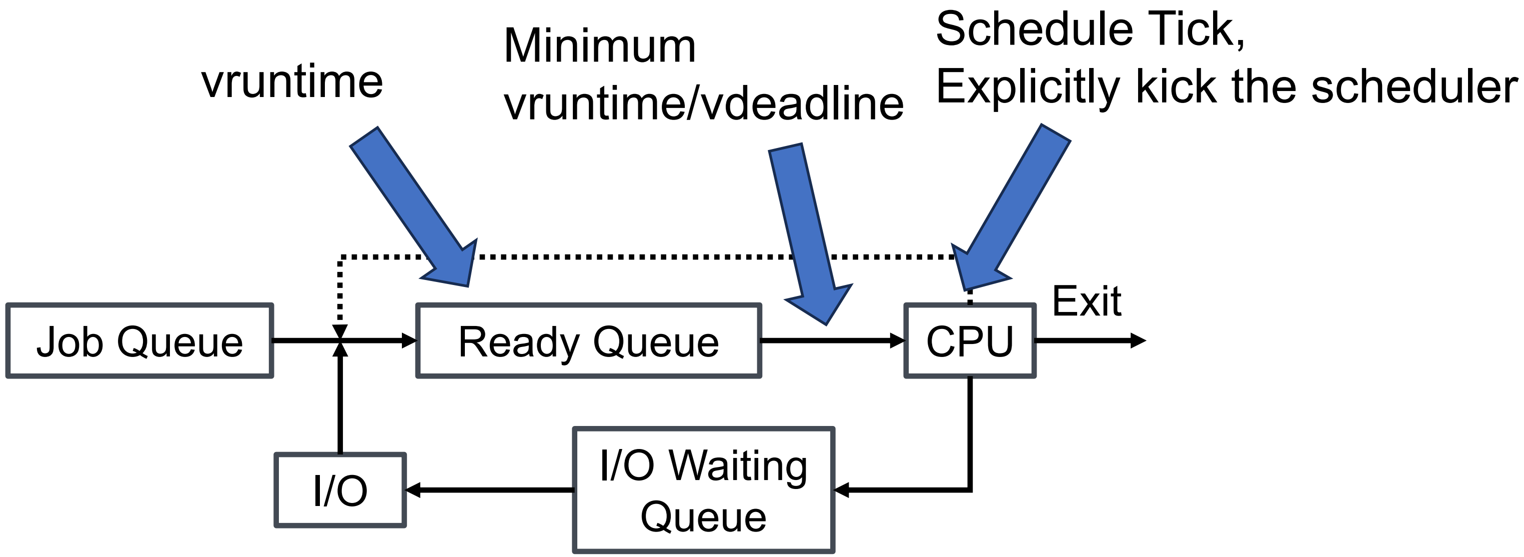
- Tasks' real execution time weighted by tasks' niceness.



CFS/EEVDF

Virtual run time (vruntime)

- Tasks' real execution time weighted by tasks' niceness.



CFS/EEVDF - Limitations

Coarse-Grained Schedule Tick vs. Strict SLOs

- Typically, the schedule tick is milliseconds.
- However, some latency-critical tasks require sub-milliseconds end-to-end response time.

Generality vs. Customization

- It is difficult to characterize tasks with a single value indicator.
- It is exhausting to develop, debug, and deploy a customized kernel scheduler.

Optimization

Coarse-Grained Schedule Tick vs. Strict SLOs

- Use dedicated cores to send fine-grained IPIs instead of coarse-grained timer interrupts. [Shinjuku]
- Use user-level libraries to support cooperative threading.

Generality vs. Customization

- Customize the scheduler policy to meet the workload properties.
- Wrap the scheduler as a kernel module. [Plugsched]
- Separate the policy and mechanism. [Enoki, sched_ext]
- Delegate the scheduling decision making to userspace. [ghOSt]

eBPF-Extensible Scheduler Class

What is the sched_ext (SCX)?

- SCX is a Linux kernel feature which enables implementing and dynamically loading safe kernel thread schedulers in BPF.

Why we need SCX?

- Small changes in scheduling behavior can have a significant impact on various components of a system.
- Use-cases have become increasingly complex and diverse.
- Experimenting with CFS directly or implementing a new sched_class from scratch is often difficult and time consuming.

eBPF

What is eBPF used for?

- Run user programs in a privileged context (e.g., kernel).
- Safely and efficiently extends the capabilities of the kernel at runtime.
- No need to change the kernel source code or load kernel modules.

eBPF Programs

Where can eBPF programs run in the kernel?

- Where a program can attach and what it is allowed to do depends on its program type.

Program Types

- Network
- cGroup
- Tracing
- Misc (e.g., struct ops, syscall)

ELF Section Names

- Libbpf uses ELF section names to convey the program type.

A Minimum eBPF Program

```c
/* SPDX-License-Identifier: (LGPL-2.1 OR BSD-2-Clause) */
#define BPF_NO_GLOBAL_DATA
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>

typedef unsigned int u32;
typedef int pid_t;
const pid_t pid_filter = 0;

char LICENSE[] SEC("license") = "Dual BSD/GPL";

SEC("tp/syscalls/sys_enter_write")
int handle_tp(void *ctx)
{
 pid_t pid = bpf_get_current_pid_tgid() >> 32;
 if (pid_filter && pid != pid_filter)
  return 0;
 bpf_printk("BPF triggered sys_enter_write from PID %d.\n", pid);
 return 0;
}
```

eBPF Verifier

How to ensure the kernel safety?

- The eBPF verifier checks the program against a set of rules.

What does the verifier do?

- Walk over each instruction and update the state of the registers and stack.
- Check every possible permutation of a program.
- Keep track of data types.
- Null checks before dereferencing pointers.

eBPF Maps

What are eBPF maps?

- Efficient key/value stores that reside in kernel space
- Communicate between a user space application and in-kernel eBPF code
- Share data amang multiple eBPF programs

```
struct {
	__uint(type, BPF_MAP_TYPE_ARRAY);
	__type(key, u32);
	__type(value, struct vpid_bitmap);
	__uint(max_entries, 1);
} vpid_bitmap_stor SEC(".maps");
```

```
struct {
	__uint(type, BPF_MAP_TYPE_TASK_STORAGE);
	__uint(map_flags, BPF_F_NO_PREALLOC);
	__type(key, s32);
	__type(value, struct task_ctx);
	__uint(pinning, LIBBPF_PIN_BY_NAME);
} task_ctx_stor SEC(".maps");
```

Take All Together

How to use eBPF?

- Import helper libraries.
- Select points we are interested in to attach eBPF programs.
- Define proper eBPF maps for communication.
- Write eBPF programs and user programs.
- Fighting the eBPF verifier.

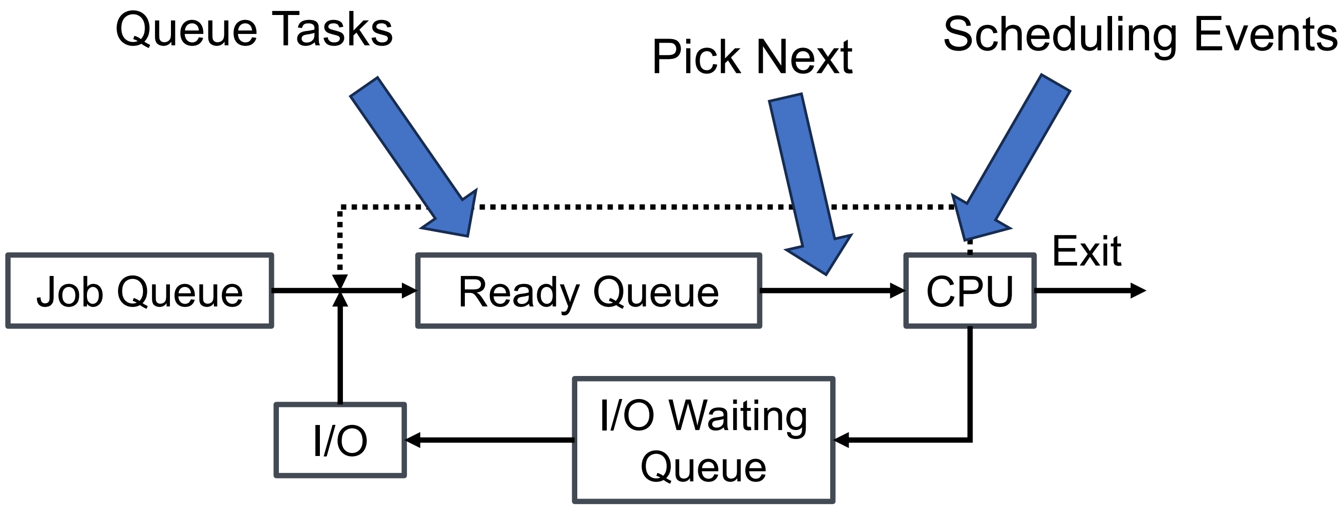
eBPF For Scheduling

How does SCX accelerate scheduler development?

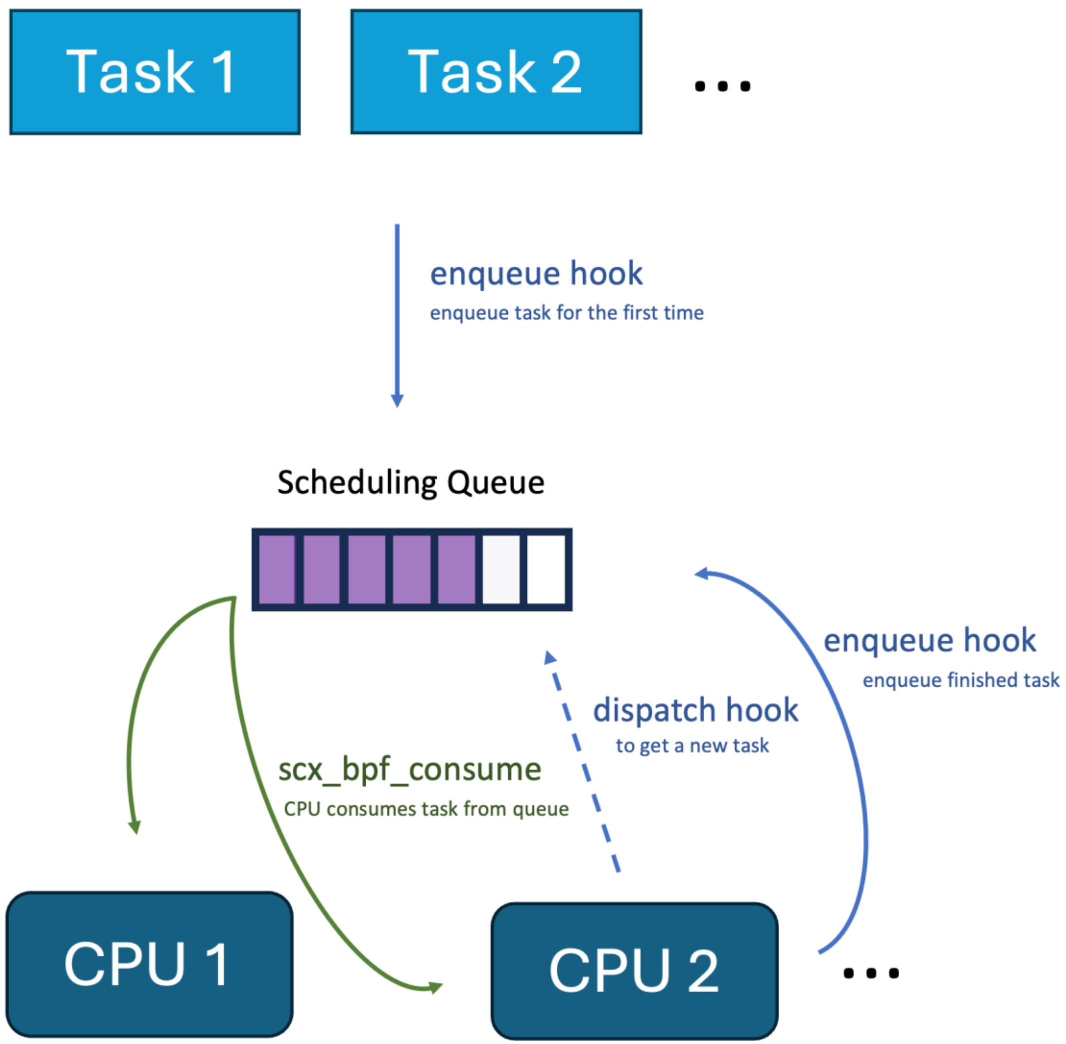
- Simplify the scheduling model and expose friendly APIs.
- Dynamically load the scheduler without reboots.
- Provide a fallback mechanism to avoid system crashes

```
                                                  +=========================+
                                                  || User-space part of    ||
                                                  || your scheduler        ||
                                                  || (e.g., main.rs)       ||
                                                  +=========================+
~~~~~~~~~~~~~~~~ <kernel space vs. user space boundary > ~~~~~~~~~~~~~~~~~~
                                                  +=========================+
                                                  || Your BPF scheduler    ||
                                                  || (e.g., main.bpf.c)    ||
                                                  +=========================+
+-----------------------+ +---------------------+ +=========================+
| Default scheduler     | | Real-time scheduler | || Sched_ext framework   ||
| (EEVDF)               | | (FIFO, RR)          | ||                       ||
| (kernel/sched/fair.c) | | (kernel/sched/rt.c) | || (kernel/sched/ext.c)  ||
+-----------------------+ +---------------------+ +=========================+
+--------------------------------------------------------------------------+
|   Core kernel scheduler                                                  |
|   (kernel/sched/core.c)                                                  |
+--------------------------------------------------------------------------+
```

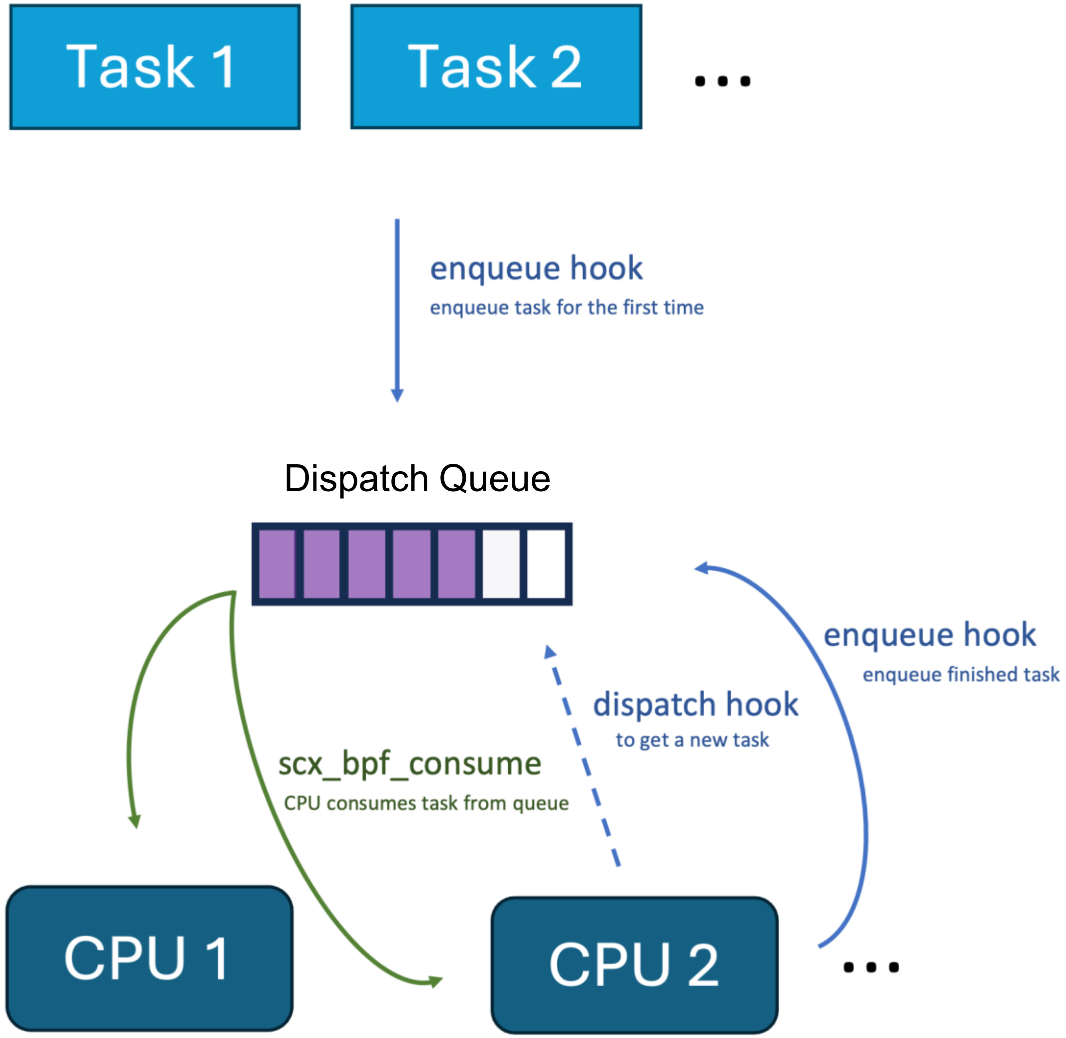
Simplified Scheduler Model



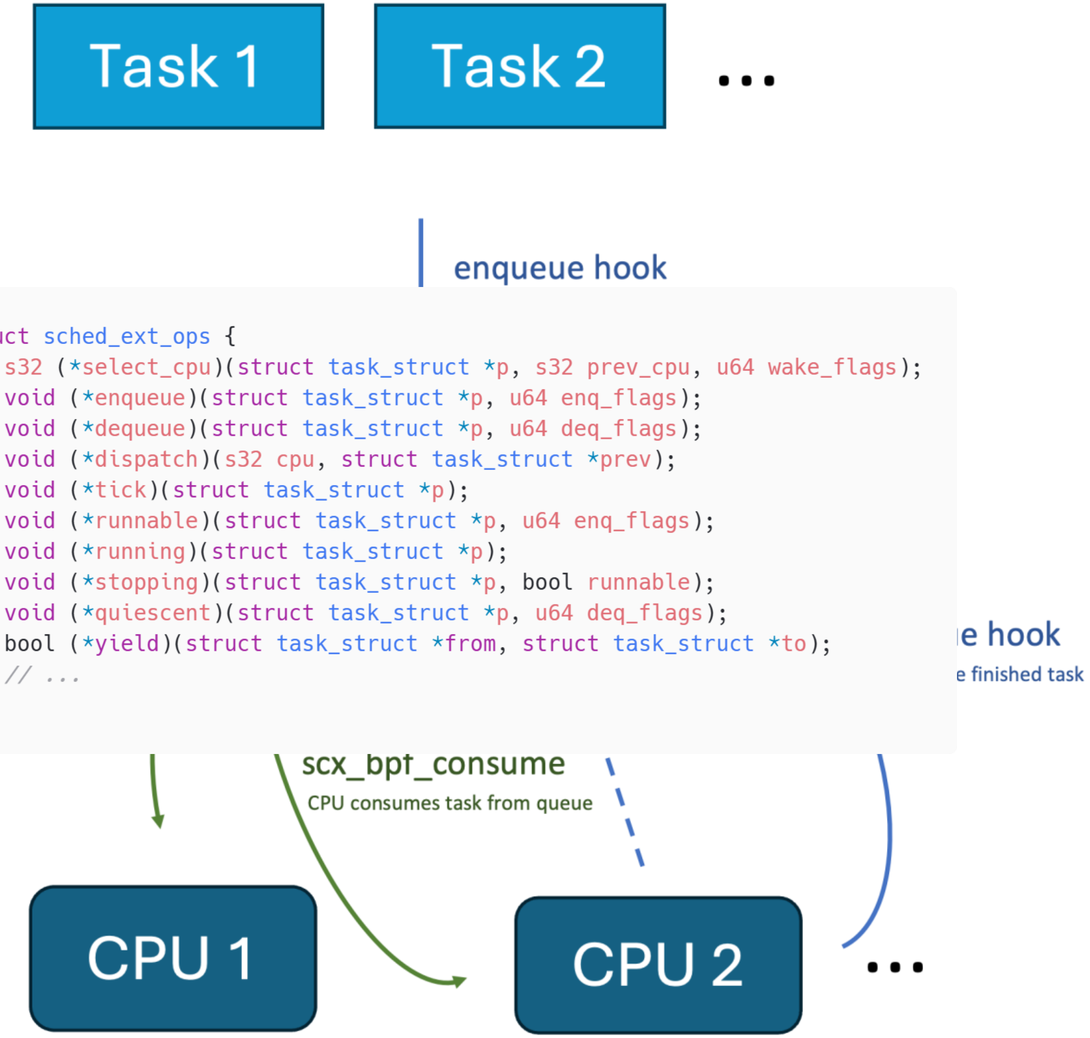
Simplified Scheduler Model



Simplified Scheduler Model



Simplified Scheduler Model




```
struct sched_ext_ops {
    s32 (*select_cpu)(struct task_struct *p, s32 prev_cpu, u64 wake_flags);
    void (*enqueue)(struct task_struct *p, u64 enq_flags);
    void (*dequeue)(struct task_struct *p, u64 deq_flags);
    void (*dispatch)(s32 cpu, struct task_struct *prev);
    void (*tick)(struct task_struct *p);
    void (*runnable)(struct task_struct *p, u64 enq_flags);
    void (*running)(struct task_struct *p);
    void (*stopping)(struct task_struct *p, bool runnable);
    void (*quiescent)(struct task_struct *p, u64 deq_flags);
    bool (*yield)(struct task_struct *from, struct task_struct *to);
    // ...
};
```

SCX Practice - BPF Maps

```
struct scx_rq_ctx {
    u32 cpu;
    struct load_weight load;
    u32 nr_running;
    s64 avg_vruntime;
    u64 avg_load;
    u64 min_vruntime;
};

struct task_ctx {
    u32 vpid;
    struct load_weight load;
    u64 deadline;
    u64 min_vruntime;
    bool on_rq;
    u64 exec_start;
    u64 sum_exec_runtime;
    u64 prev_sum_exec_runtime;
    u64 vruntime;
    s64 vlag;
    u64 slice;
};
```

```
struct {
    __uint(type, BPF_MAP_TYPE_PERCPU_ARRAY);
    __type(key, u32);
    __type(value, struct scx_rq_ctx);
    __uint(max_entries, 1);
} scx_rq_ctx_stor SEC(".maps");

struct {
    __uint(type, BPF_MAP_TYPE_TASK_STORAGE);
    __uint(map_flags, BPF_F_NO_PREALLOC);
    __type(key, s32);
    __type(value, struct task_ctx);
} task_ctx_stor SEC(".maps");
```

SCX Practice - BPF Programs

```c
void BPF_STRUCT_OPS(eevdf_enqueue, struct task_struct *p, u64 enq_flags)
{
    s32 cpu = scx_bpf_task_cpu(p);

    struct scx_rq_ctx *qctx = try_lookup_scx_rq_ctx(cpu);
    if (unlikely(!qctx))
        return;
    struct task_ctx *tctx = try_lookup_task_ctx(p);
    if (unlikely(!tctx))
        return;

    update_curr(qctx);
    place_task(qctx, tctx, 0);
    update_enq(qctx, tctx);

    struct task_struct *curr = curr_task(qctx);
    if (curr && task_nice(p) < task_nice(curr)) {
        scx_bpf_dsq_insert(p, SCX_DSQ_LOCAL, SCX_SLICE_DFL,
                   enq_flags | SCX_ENQ_PREEMPT);
        return;
    }

    if (task_eligible(qctx, tctx))
        scx_bpf_dsq_insert_vtime(p, eligible_dsq(cpu), SCX_SLICE_DFL,
                     tctx->deadline, enq_flags);
    else
        scx_bpf_dsq_insert_vtime(p, ineligible_dsq(cpu), SCX_SLICE_DFL,
                     tctx->vlag, enq_flags);
}
```

```c
void BPF_STRUCT_OPS(eevdf_dequeue, struct task_struct *p, u64 deq_flags)
{
    struct scx_rq_ctx *qctx = try_lookup_scx_rq_ctx(scx_bpf_task_cpu(p));
    if (unlikely(!qctx))
        return;
    struct task_ctx *tctx = try_lookup_task_ctx(p);
    if (unlikely(!tctx))
        return;

    update_curr(qctx);
    update_task_lag(qctx, tctx);
    update_deq(qctx, tctx);
}

void BPF_STRUCT_OPS(eevdf_dispatch, s32 cpu, struct task_struct *prev)
{
    struct scx_rq_ctx *qctx = try_lookup_scx_rq_ctx(cpu);
    if (unlikely(!qctx))
        return;

    struct task_struct *p = pick_first_task(qctx);
    if (!p || p == prev)
        return;

    struct task_ctx *tctx = try_lookup_task_ctx(p);
    if (unlikely(!tctx))
        return;

    scx_bpf_dsq_move_to_local(p->scx.dsq->id);
}
```

SCX Practice

```c
void BPF_STRUCT_OPS(eevdf_tick, struct task_struct *p)
{
    bool resched = false;

    struct scx_rq_ctx *qctx = this_scx_rq_ctx();
    if (unlikely(!qctx))
        return;

    struct task_ctx *tctx = try_lookup_task_ctx(p);
    if (unlikely(!tctx))
        return;

    resched = update_curr(qctx);

    u64 now = scx_bpf_now();
    if (unlikely(!now)) {
        scx_bpf_error("Invalid time!");
        return;
    }
    if (resched || !tctx->slice) {
        p->scx.slice = 0;
        // p->scx.core_sched_at = now;
    }
}
```

```c
SCX_OPS_DEFINE(eevdf_ops,
           .select_cpu = (void *)eevdf_select_cpu, //
           .enqueue = (void *)eevdf_enqueue, //
           .dequeue = (void *)eevdf_dequeue, //
           .dispatch = (void *)eevdf_dispatch, //
           .dispatch_max_batch = 1, //
           .tick = (void *)eevdf_tick, //
           .running = (void *)eevdf_running, //
           .stopping = (void *)eevdf_stopping, //
           .enable = (void *)eevdf_enable, //
           .set_weight = (void *)eevdf_set_weight, //
           .cpu_release = (void *)eevdf_cpu_release, //
           .init_task = (void *)eevdf_init_task, //
           .exit_task = (void *)eevdf_exit_task, //
           .init = (void *)eevdf_init, //
           .exit = (void *)eevdf_exit, //
           .name = "eevdf");
```

Network I/O Scheduling

Why is network scheduling special?

- Large-scale, datacenter applications pose unique challenges to system software and their network stack in two aspects:

  ### Microsecond Tail Latency

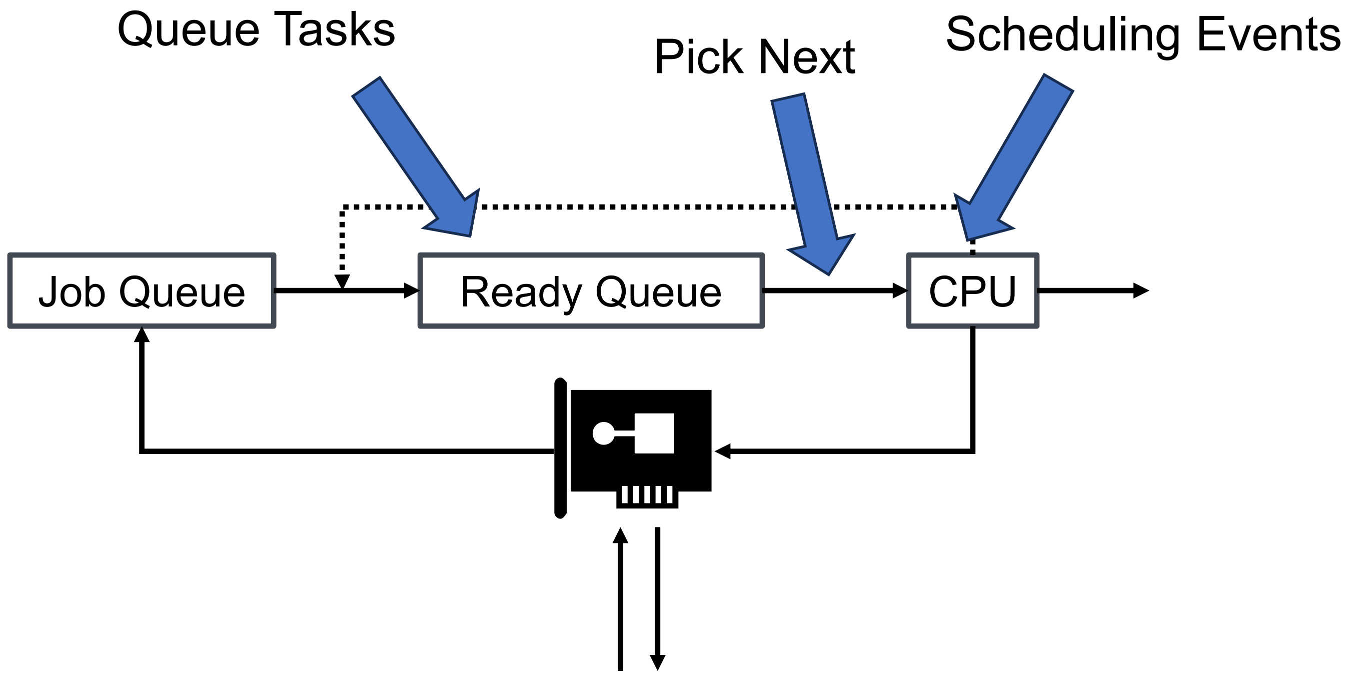
  - Each user request often involves hundreds of servers, and the end-to-end response time highly associates with the slowest server.
  - The system network stack plays a significant role in exacerbating the problem.

  ### High Package Rates

  - Most service request packages are quite small (hundreds of bytes).
  - Each node can scale to serve millions of requests per second.

The Hardware-OS Mismatch

- Multiple applications share a single processing core.
- Packet interarrival times much higher than the latency of interrupts and system calls.

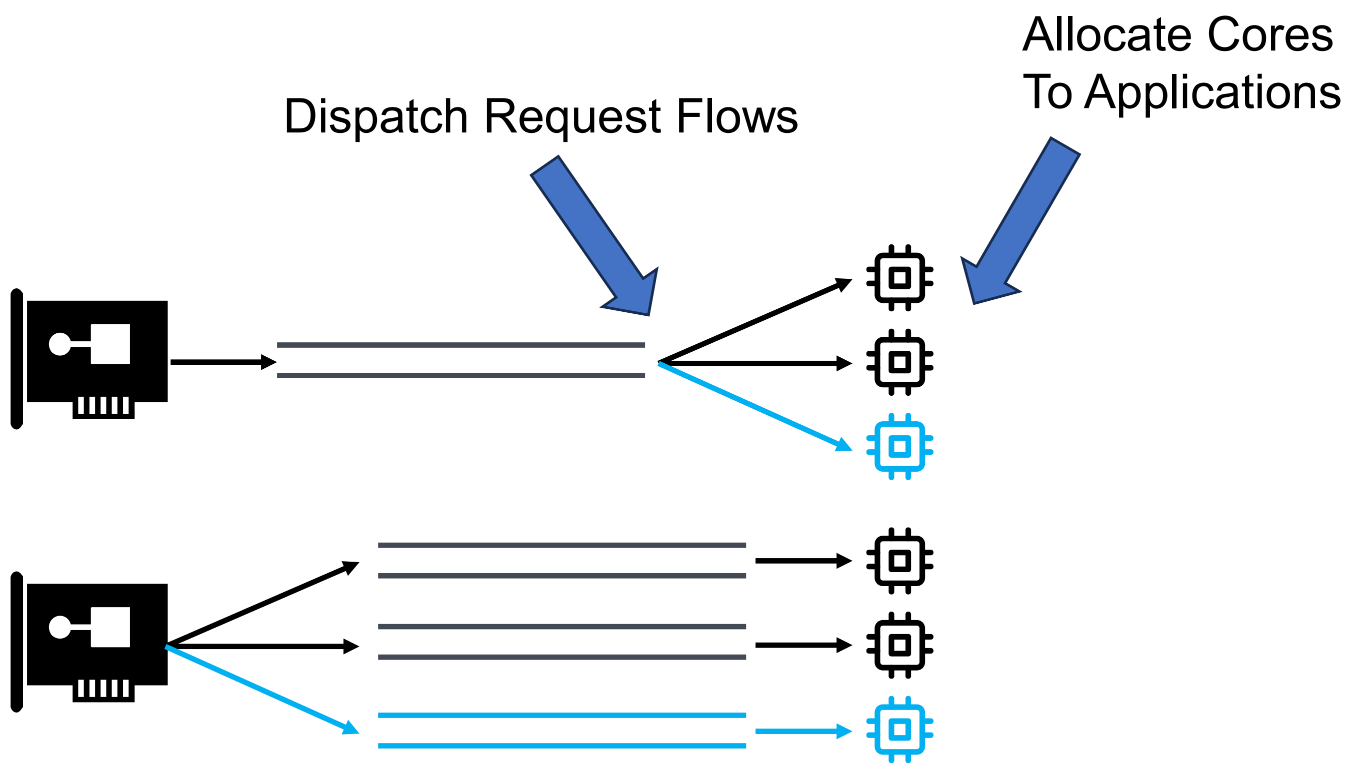


Optimization

- Separate dataplane and control plane
- Zero copy
- Run-to-completion
- Synchronization-free, flow-consistent mapping of requests to cores
- Multi-queue NIC, RSS

The Best Policy Is No Policy

- FCFS
- Do not share cores.



FCFS - Challenges

Difficult to balance workloads.

- High-variance dispersed workload distribution
- Head-of-line blocking
- Work conserving

Difficult to pre-/re-allocate resources.

- Bounded/unbounded workloads
- Clairvoyant/non-clairvoyant scheduling

Balance Workloads

High-variance dispersed workload distribution

- Leave some cores for burst [Perséphone]

Head-of-line blocking

- Processor Sharing
- Preemption [Shinjuku]

Work conserving

- Work steeling [ZygOS]
- Dynamically allocate resource [Shenago]

Pre-/Re-allocate Resources

- Learn request flow characteristics by ML
- Approximize lower boundary/upper boundary [QCLIMB]
- Coordinate with centralized dispatcher [Junction]

Summary

- Brief intro to scheduling
- CPU/thread scheduling
- eBPF
- Develop a scheduler with sched_ext
- Network scheduling concepts

Thanks for listening