Learning sched_ext: BPF extensible scheduler class …

@shun159

Disclaimer

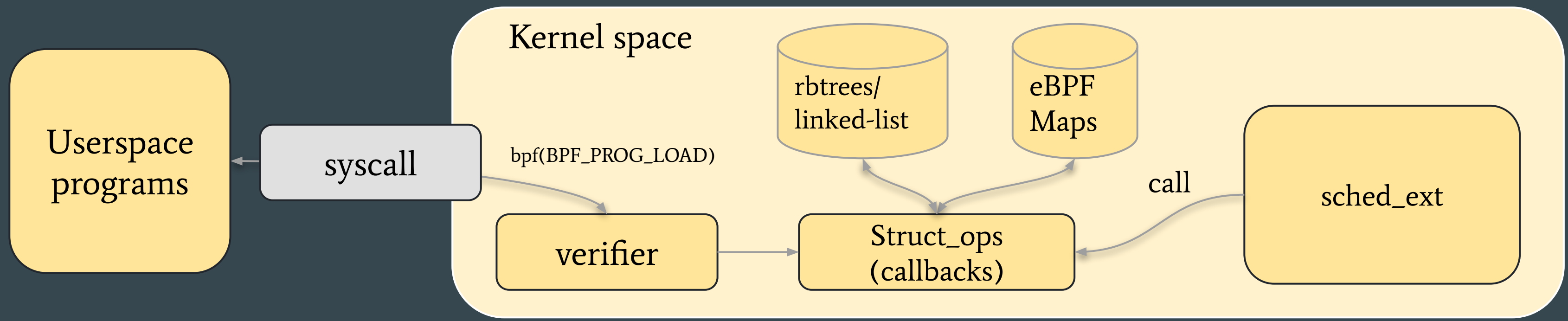
- I’m a newbie at Linux kernel
    - And also, I’m not a scheduler expert.

Introduction

- sched_ext: new extensible scheduler class
    - Allows scheduling policies to be implemented as BPF programs
    - Provides simple and intuitive API for implement policies
        - Doesn’t require knowledge of core scheduler internals
    - Allows that experimentation in a safe manner without even needing to reboot the system
        - Safe, cannot crash the host
            - Protection afforded by BPF verifier
    - Used in Meta production to optimize their workloads

Implementing scheduling policies: Overview

- Userspace can implement an arbitrary CPU scheduler by loading a BPF programs that implement "sched_ext_ops"
- BPF program must implement a set of callbacks
  - Task wakeup
  - Task enqueue/dequeue
  - Task state change (runnable, running, stopping, quiescent)
  - ...
- Like other eBPF programs, we can use eBPF maps/data structures as needed



Implementing scheduling policies: Callbacks(1)

```
/* Pick the target CPU for a task which is being woken up */
s32 (*select_cpu)(struct task_struct *p, s32 prev_cpu, u64 wake_flags);

/* Enqueue a runnable task on the BPF scheduler or dispatch directly to CPU */
void (*enqueue)(struct task_struct *p, u64 enq_flags);

/* Remove a task from the BPF scheduler.
 *  This is usually called to isolate the task while updating its scheduling properties (e.g. priority). */
void (*dequeue)(struct task_struct *p, u64 deq_flags);
....
/* BPF scheduler’s name, 128 chars or less */
char name[SCX_OPS_NAME_LEN];
```

Implementing scheduling policies: Callbacks(2)

```
/* A task is becoming runnable on its associated CPU */
void (*runnable)(struct task_struct *p, u64 enq_flags);

/* A task is starting to run on its associated CPU */
void (*running)(struct task_struct *p);

/* A task is starting to run on its associated CPU */
void (*stopping)(struct task_struct *p, bool runnable);

/* A task is becoming not runnable on its associated CPU */
void (*quiescent)(struct task_struct *p, u64 deq_flags);
```

<u>The only thing we need to implement is the “name” of the scheduler; everything else is optional</u>

Implementing scheduling policies: BPF program

```
s32 BPF_STRUCT_OPS(simple_init)
{
        if (!switch_partial)
                scx_bpf_switch_all();
        return 0;
}
```

All existing and future CFS tasks(**SCHED_NORMAL, SCHED_BATCH, SCHED_IDLE and SCHED_EXT**) switched to SCX. Otherwise, only tasks that have **SCHED_EXT** explicitly set will be placed on sched_ext.

```
void BPF_STRUCT_OPS(simple_enqueue, struct task_struct *p, u64 enq_flags)
{
        if (enq_flags & SCX_ENQ_LOCAL)
                scx_bpf_dispatch(p, SCX_DSQ_LOCAL, SCX_SLICE_DFL, enq_flags);
        else
                scx_bpf_dispatch(p, SCX_DSQ_GLOBAL, SCX_SLICE_DFL, enq_flags);
}
```

When **SCX_ENQ_LOCAL** is set in the enq_flag, it indicates that running the task on the selected CPU directly should not affect fairness. In this case, just queue it on the local FIFO.

Otherwise, in this example code, re-enqueue the task directly in the global DSQ. It will be consumed later by sched_ext.

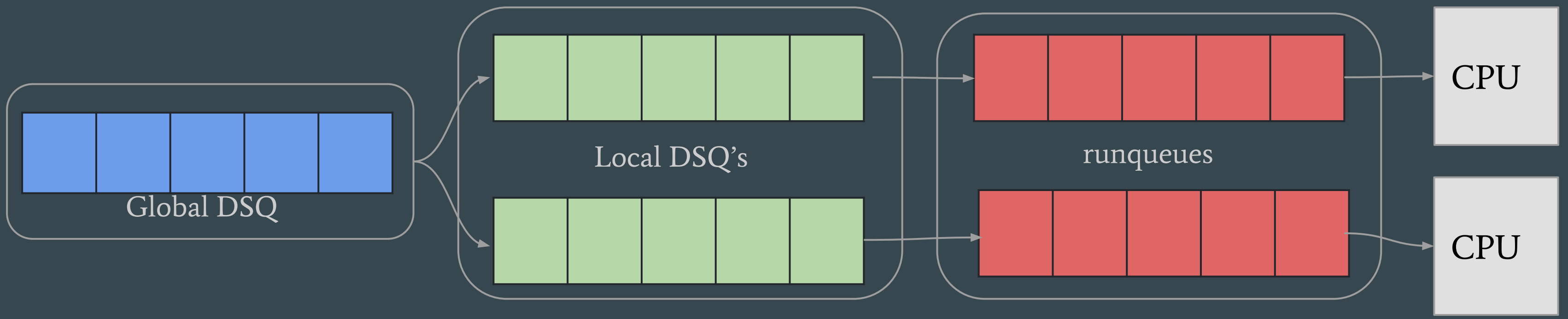
```
void BPF_STRUCT_OPS(simple_exit, struct scx_exit_info *ei)
{
        exit_type = ei->type;
}

SEC(".struct_ops")
struct sched_ext_ops simple_ops = {
        .enqueue                = (void *)simple_enqueue,
        .init                   = (void *)simple_init,
        .exit                   = (void *)simple_exit,
        .name                   = "simple",
};
```

Specify function pointers that called by sched_ext.

DSQ’s(Dispatch Queues): Overview

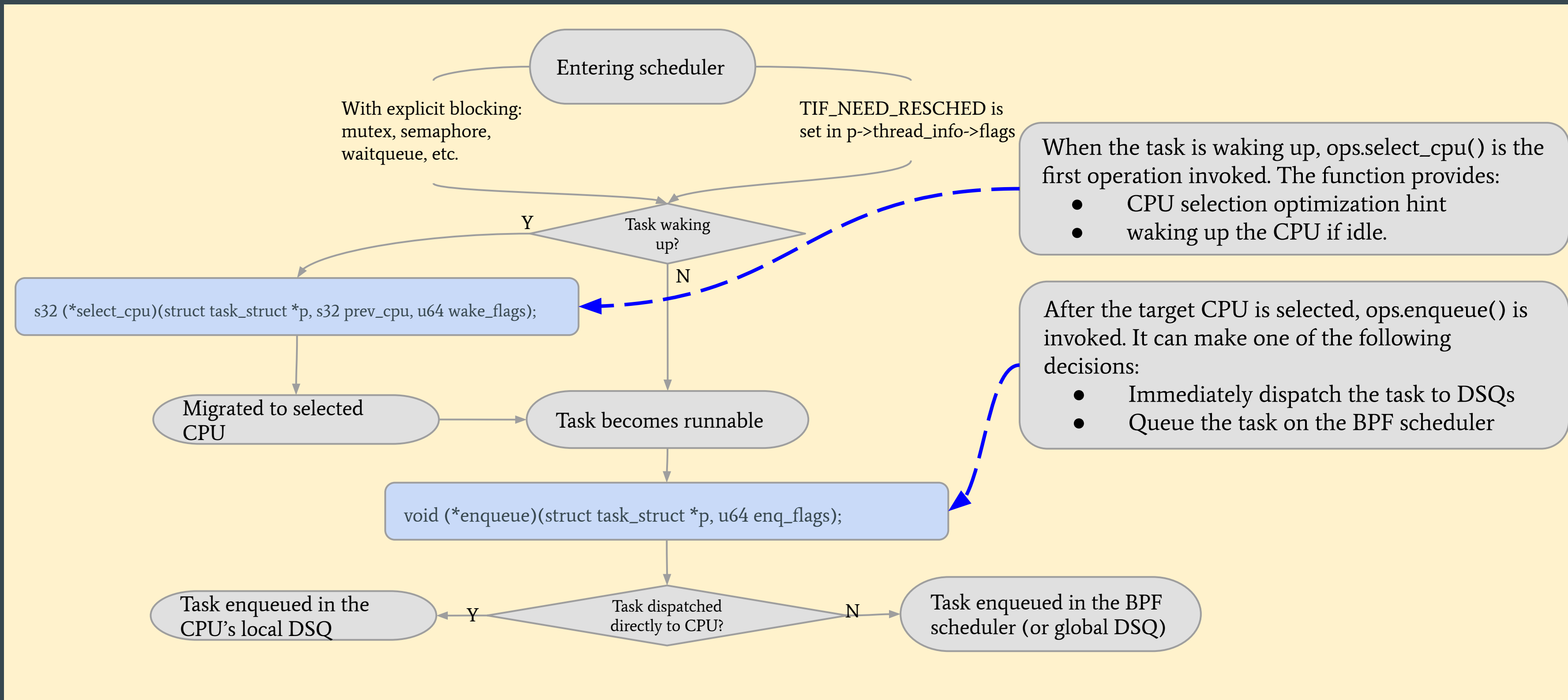
- An abstraction layer between BPF scheduler and kernel for managing queues of tasks, sched_ext uses a FIFO queue called DSQ’s(Dispatch Queues).
  - By default, one global DSQ and a per-CPU local DSQ are created.
  - Global DSQ (SCX_DSQ_GLOBAL)
    - By default, consumed when the local DSQs are empty.
    - Can be utilized by a scheduler if necessary
  - per-CPU local DSQ’s (SCX_DSQ_LOCAL)
    - per-CPU FIFO that SCX pulls from when putting the next task on the CPU.
    - A CPU always executes a task from its local DSQ



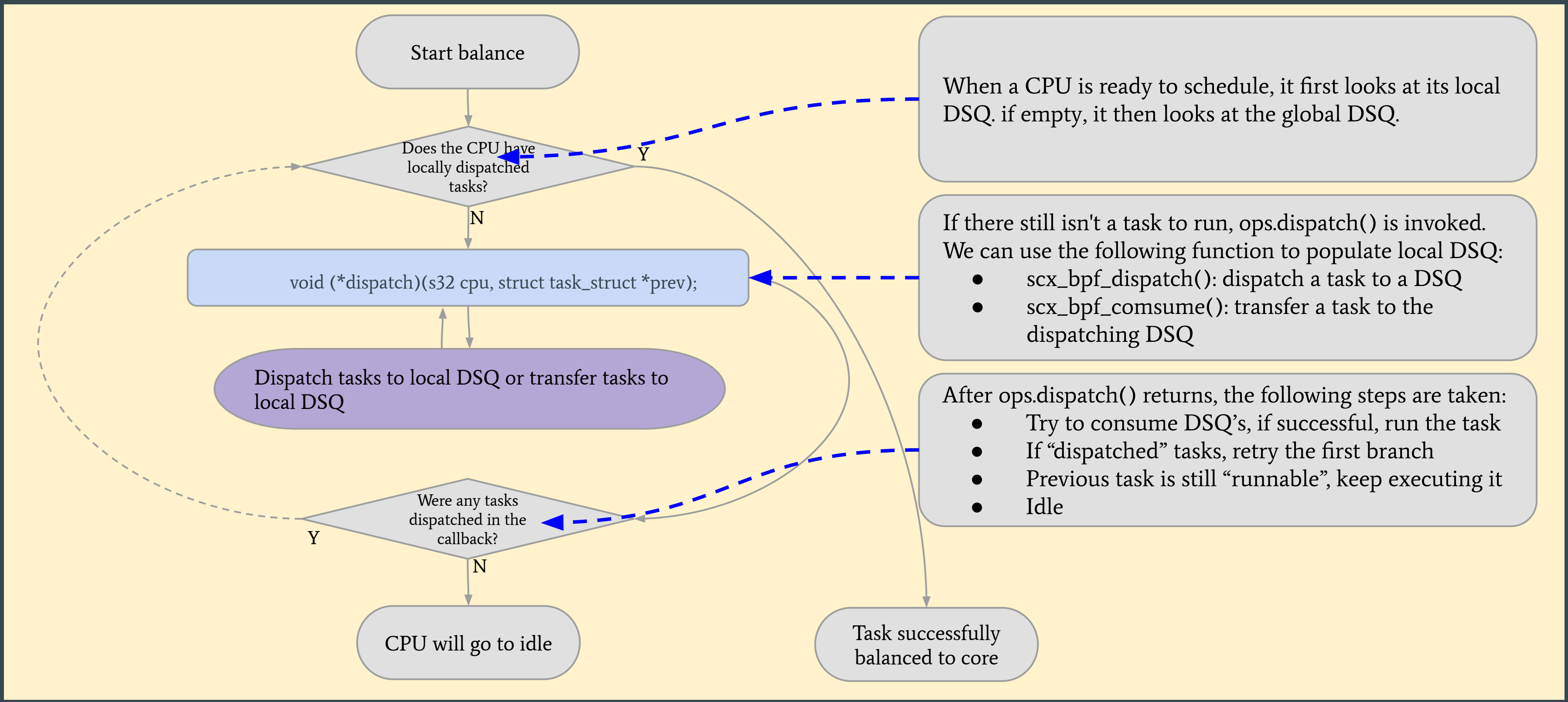
DSQ's(Dispatch Queues): Operations

- Each DSQ provides operations; "dispatch" and "consume"
  - "Consume":
    - Like as pick_next_task(), consuming a next task from a DSQ to run on the calling CPU.
    - consumed in ops.dispatch() when a core is will go idle if no task is found
  - "Dispatch": Placing a task into a CPU. Can be done in the following callbacks
    - ops.enqueue: invoked when a task is being enqueued in the BPF scheduler.
    - ops.dispatch: invoked when a CPU is will go idle if a task is not found.
    - This operation should either dispatch one or more tasks to other local DSQs or transfer a task from a DSQ to the current CPU's DSQ

Scheduling Cycle: Task enqueue/wakeup flow



Scheduling Cycle: Runqueue balance/dispatch



Build sched-ext kernel (1)

1. Checkout the sched_ext repo from github:

```
git clone https://github.com/sched-ext/sched_ext
```

2. Checkout and build the latest clang:

```
$ yay -S cmake ninja
$ mkdir ~/llvm
$ git clone https://github.com/llvm/llvm-project.git llvm-project
$ mkdir -p llvm-project/build; cd llvm-project/build
$ cmake -G Ninja \
    -DLLVM_TARGETS_TO_BUILD="BPF;X86" \
    -DCMAKE_INSTALL_PREFIX="/$HOME/llvm/$(date +%Y%m%d)" \
    -DBUILD_SHARED_LIBS=OFF \
    -DLIBCLANG_BUILD_STATIC=ON \
    -DCMAKE_BUILD_TYPE=Release \
    -DLLVM_ENABLE_TERMINFO=OFF \
    -DLLVM_ENABLE_PROJECTS="clang;lld" \
    ../llvm
$ ninja install -j$(nproc)
$ ln -sf /$HOME/llvm/$(date +%Y%m%d) /$HOME/llvm/latest
```

Build sched-ext kernel (2)

3. Download and build the latest pahole:

```
$ cd /data/users/$USER
$ git clone https://git.kernel.org/pub/scm/devel/pahole/pahole.git
$ mkdir -p pahole/build; cd pahole/build
$ cmake -G Ninja ../ $ ninja
```

*** After build pahole and clang, make sure they are in your $PATH ***

4. Build sched_ext kernel:

CONFIG_DEBUG_INFO_DWARF_TOOLCHAIN_DEFAULT=y CONFIG_DEBUG_INFO_BTF=y CONFIG_PAHOLE_HAS_SPLIT_BTF=y CONFIG_PAHOLE_HAS_BTF_TAG=y CONFIG_SCHED_CLASS_EXT=y CONFIG_SCHED_DEBUG=y CONFIG_BPF_SYSCALL=y CONFIG_BPF_JIT=y	### 9P_FS is used by osandov-linux to mount the custom build directory from the hostmachine CONFIG_9P_FS=y CONFIG_NET_9P=y CONFIG_NET_9P_FD=y CONFIG_NET_9P_VIRTIO=y

Build sched-ext kernel (3)

4. Build sched_ext kernel:

```
$ make CC=clang-17 LD=ld.lld LLVM=1 menuconfig
$ make CC=clang-17 LD=ld.lld LLVM=1 olddefconfig
$ make CC=clang-17 LD=ld.lld LLVM=1 -j$(nproc)
```

5. Build scx samples:

```
$ cd tools/sched_ext
$ make CC=clang-17 LD=ld.lld LLVM=1 -j$(nproc)
```

Build sched-ext kernel (4)

6. Setup a VM for the sched_ext kernel

I recommend using osantov-linux[0], as it is a very handy tool for running a custom-built kernel

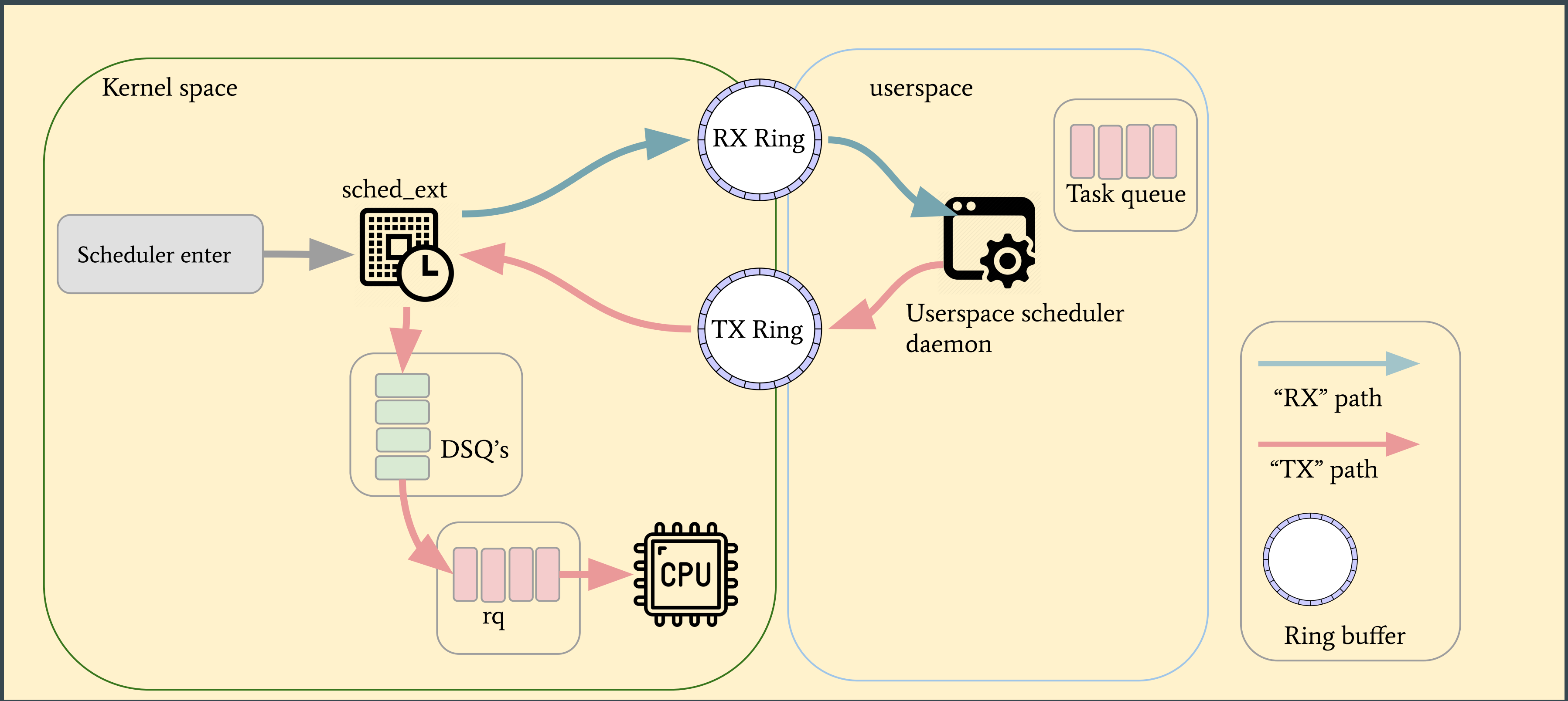
```
$ vm.py create -c 4 -m 8192 -s 50G <vm name>
$ vm.py archinstall <vm name>
$ kconfig.py <path to osandov-linux>/configs/vmpy.fragment
$ vm.py run -k $PWD -- <vm name>
```

[0]: https://github.com/osandov/osandov-linux

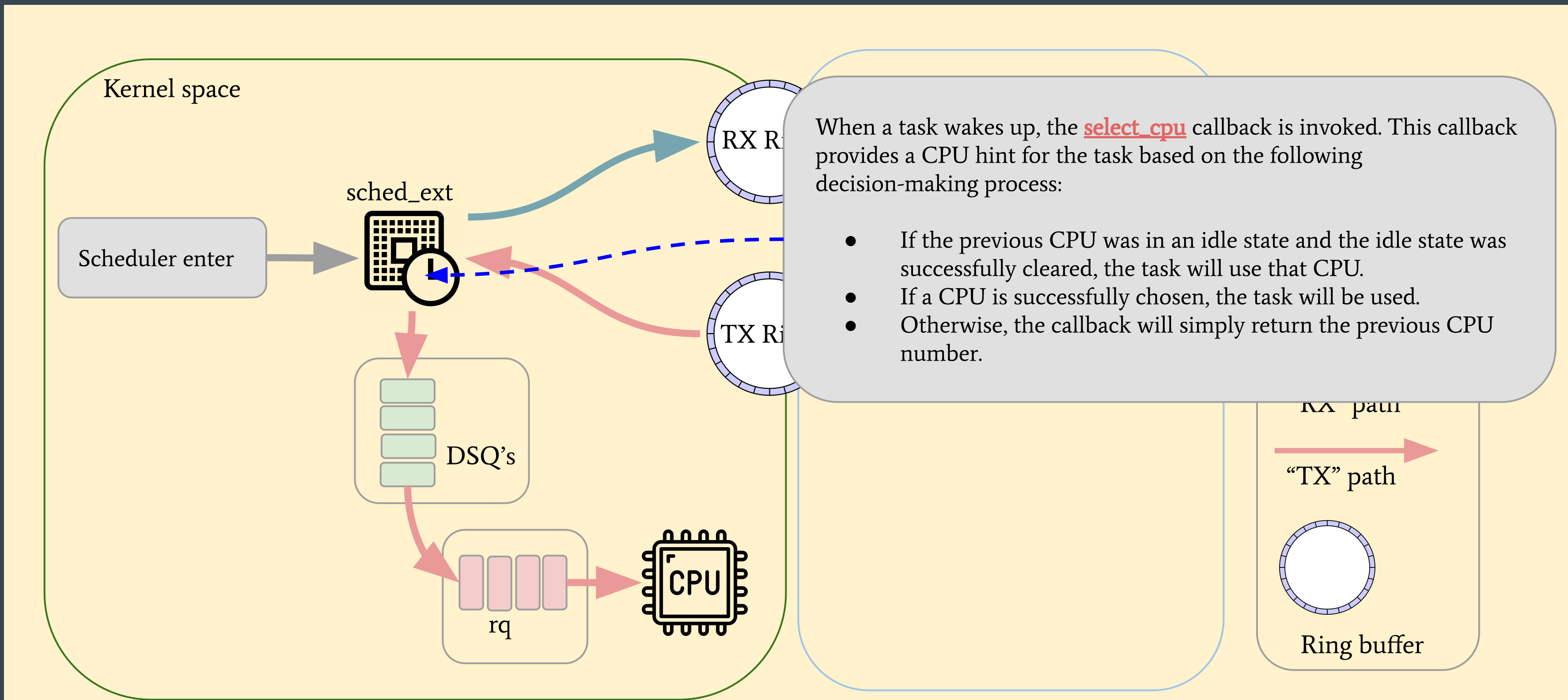
Write own CPU scheduler

- A "vruntime" scheduler that performs scheduling decisions in userspace.
- While it may not be the most practical approach, the intention is to see how write a userspace scheduler
- The sample scheduler shows how to bypass kernel scheduling tasks and handle them in userspace.
    - Move the complexity of scheduling tasks from the kernel to userspace.

Write own CPU scheduler: diagram



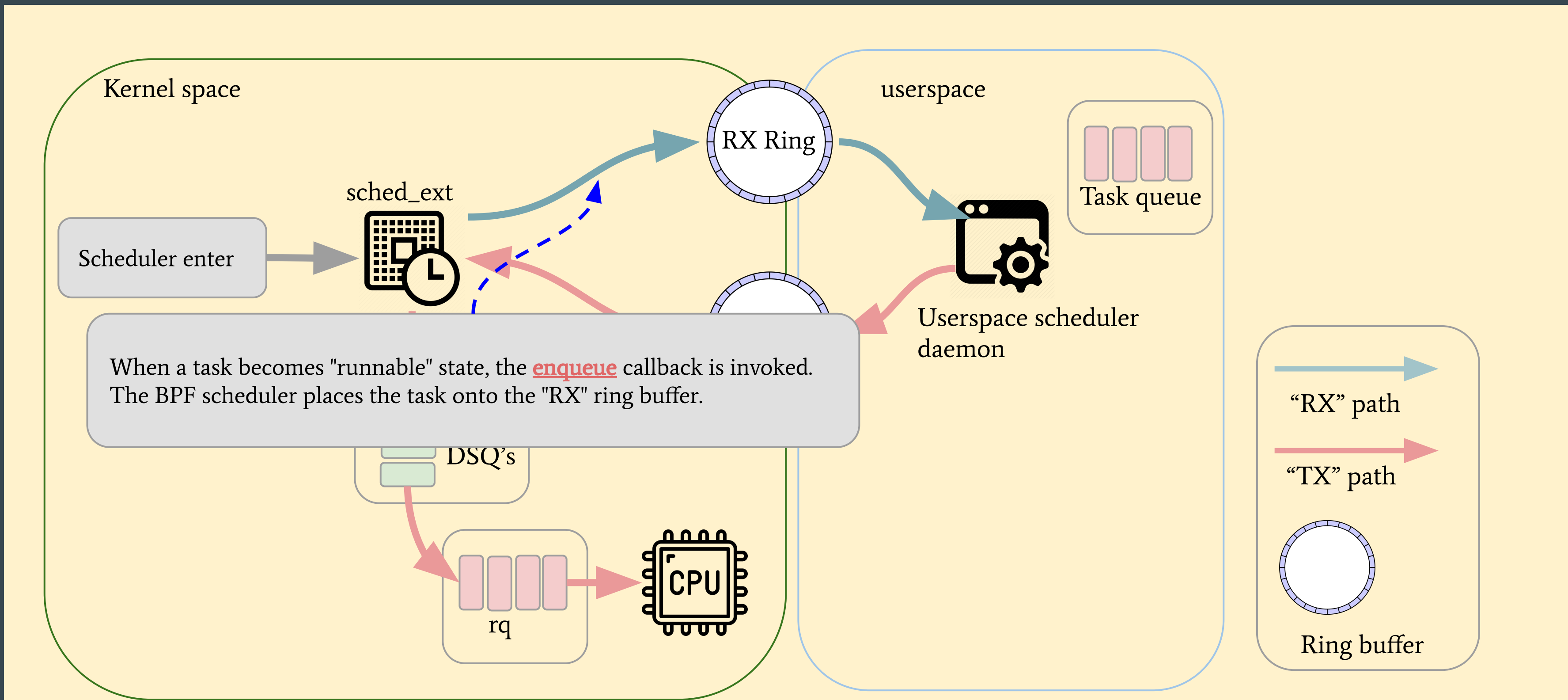
Write own CPU scheduler: Select CPU



When a task wakes up, the select_cpu callback is invoked. This callback provides a CPU hint for the task based on the following decision-making process:

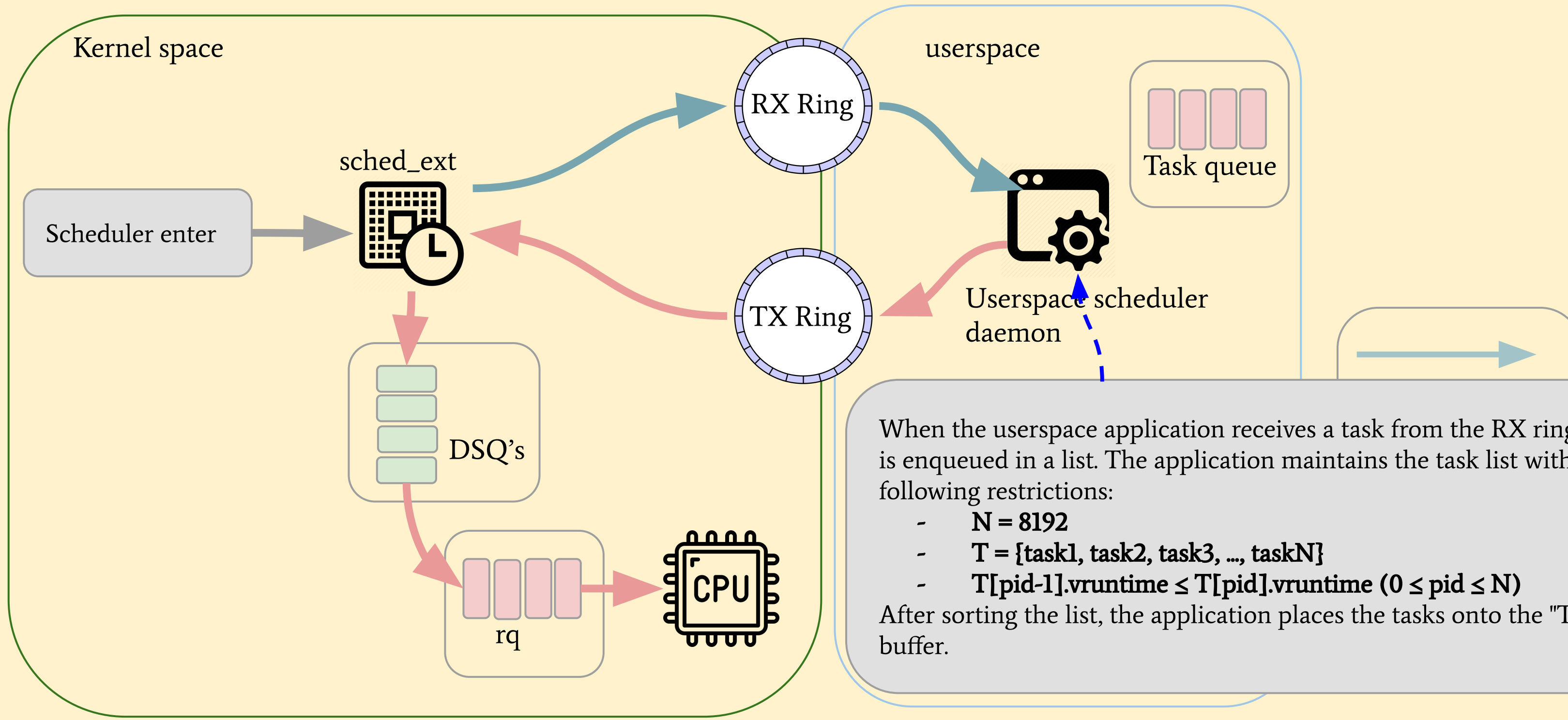
- If the previous CPU was in an idle state and the idle state was successfully cleared, the task will use that CPU.
- If a CPU is successfully chosen, the task will be used.
- Otherwise, the callback will simply return the previous CPU number.

Write own CPU scheduler: Enqueue a task



When a task becomes "runnable" state, the enqueue callback is invoked.
The BPF scheduler places the task onto the "RX" ring buffer.

Write own CPU scheduler: Enqueue a task

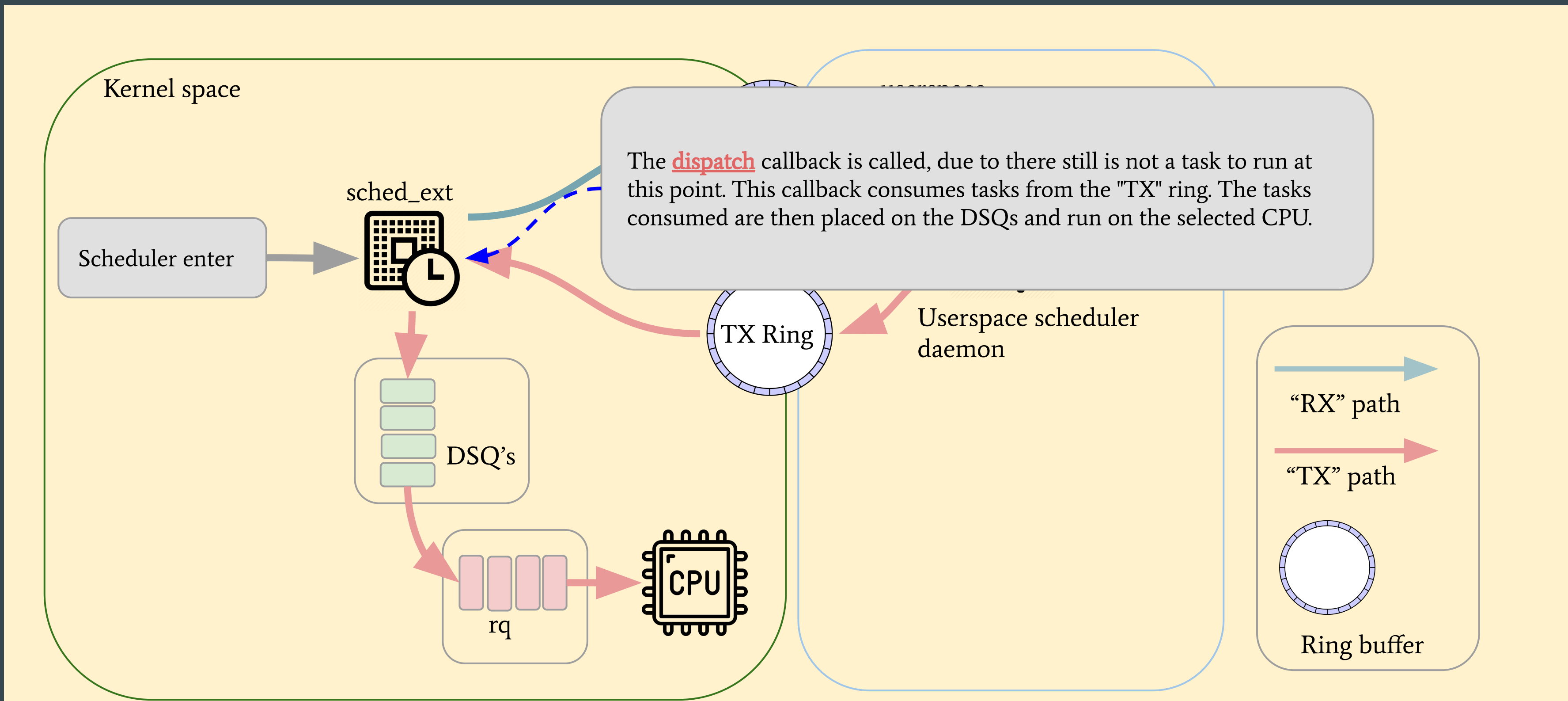


When the userspace application receives a task from the RX ring, the task is enqueued in a list. The application maintains the task list with the following restrictions:

- **N = 8192**
- **T = {task1, task2, task3, ..., taskN}**
- **T[pid-1].vruntime ≤ T[pid].vruntime (0 ≤ pid ≤ N)**

After sorting the list, the application places the tasks onto the "TX" ring buffer.

Write own CPU scheduler: Dispatch tasks



Thank you