```
/*****************************************************************
 * HFSC Scheduler – Hierarchical + Heap + Rule Table
 * DPDK Dataplane Implementation (Paper-Faithful)
 *****************************************************************/

#include <stdint.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>

#include <rte_mbuf.h>
#include <rte_ring.h>
#include <rte_cycles.h>
#include <rte_ip.h>
#include <rte_tcp.h>
#include <rte_udp.h>
#include <rte_ether.h>

/* ================= CONFIG ================= */

#define QUEUE_SIZE   8192
#define MAX_RT_HEAP  16   /* only leaf classes */

/* ================= SERVICE CURVE ================= */

typedef struct {
```

```
    uint64_t m1;   /* bytes/sec */
    uint64_t m2;
    uint64_t x;
} service_curve_t;

/* ================= HFSC CLASS ================= */

typedef struct hfsc_class {
    struct hfsc_class *parent;
    struct hfsc_class **children;
    uint8_t num_children;

    struct rte_ring *q;   /* only for leaf */

    service_curve_t sc;

    uint64_t eligible_time;
    uint64_t deadline;
    uint64_t virtual_time;

    bool is_leaf;
    bool active;
} hfsc_class_t;

/* ================= RT HEAP ================= */
```

```
static hfsc_class_t *rt_heap[MAX_RT_HEAP];
static int rt_heap_sz = 0;

/* ---------- Heap helpers ---------- */

static inline bool
rt_less(hfsc_class_t *a, hfsc_class_t *b)
{
    if (a->eligible_time != b->eligible_time)
        return a->eligible_time < b->eligible_time;
    return a->deadline < b->deadline;
}

static void
rt_heap_push(hfsc_class_t *cl)
{
    int i = rt_heap_sz++;
    rt_heap[i] = cl;

    while (i > 0) {
        int p = (i - 1) / 2;
        if (rt_less(rt_heap[p], rt_heap[i]))
            break;
        hfsc_class_t *tmp = rt_heap[p];
        rt_heap[p] = rt_heap[i];
        rt_heap[i] = tmp;
```

```
        i = p;
    }
}

static void
rt_heap_remove(hfsc_class_t *cl)
{
    int i;
    for (i = 0; i < rt_heap_sz; i++)
        if (rt_heap[i] == cl)
            break;
    if (i == rt_heap_sz)
        return;

    rt_heap[i] = rt_heap[--rt_heap_sz];

    while (1) {
        int l = 2*i+1, r = 2*i+2, s = i;
        if (l < rt_heap_sz && rt_less(rt_heap[l], rt_heap[s])) s = l;
        if (r < rt_heap_sz && rt_less(rt_heap[r], rt_heap[s])) s = r;
        if (s == i) break;
        hfsc_class_t *tmp = rt_heap[s];
        rt_heap[s] = rt_heap[i];
        rt_heap[i] = tmp;
        i = s;
    }
```

```
}

static inline hfsc_class_t *
rt_heap_peek(void)
{
   return rt_heap_sz ? rt_heap[0] : NULL;
}

/* ================= GLOBAL CLASSES ================= */

static hfsc_class_t *root;
static hfsc_class_t *site1;
static hfsc_class_t *site2;

static hfsc_class_t *site1_udp;
static hfsc_class_t *site1_tcp;
static hfsc_class_t *site2_udp;
static hfsc_class_t *site2_tcp;

/* ================= TIME ================= */

static inline uint64_t now_cycles(void)
{
   return rte_get_tsc_cycles();
}
```

```
static inline uint64_t
bytes_to_cycles(uint64_t bytes, uint64_t rate)
{
    return (bytes * rte_get_tsc_hz()) / rate;
}

/* ================= ACTIVATION ================= */

static void
hfsc_activate(hfsc_class_t *cl, uint64_t now)
{
    cl->active = true;
    cl->eligible_time = now;
    cl->deadline = now;
    cl->virtual_time = cl->parent->virtual_time;

    if (cl->is_leaf)
        rt_heap_push(cl);

    if (cl->parent && !cl->parent->active)
        hfsc_activate(cl->parent, now);
}

/* ================= CLASSIFICATION ================= */

static hfsc_class_t *
```

```
hfsc_classify(struct rte_mbuf *m)
{
    struct rte_ipv4_hdr *ip =
        rte_pktmbuf_mtod_offset(
            m, struct rte_ipv4_hdr *,
            sizeof(struct rte_ether_hdr));

    if (ip->src_addr != RTE_IPV4(192,168,2,20) ||
        ip->dst_addr != RTE_IPV4(192,168,2,30))
        return NULL;

    if (ip->next_proto_id == IPPROTO_UDP) {
        struct rte_udp_hdr *udp =
            (struct rte_udp_hdr *)(ip + 1);
        uint16_t dport = rte_be_to_cpu_16(udp->dst_port);

        if (dport == 5001) return site1_udp;
        if (dport == 6001) return site2_udp;
    }

    if (ip->next_proto_id == IPPROTO_TCP) {
        struct rte_tcp_hdr *tcp =
            (struct rte_tcp_hdr *)(ip + 1);
        uint16_t dport = rte_be_to_cpu_16(tcp->dst_port);

        if (dport == 5002) return site1_tcp;
```

```
        if (dport == 6002) return site2_tcp;
    }

    return NULL;
}

/* ================= ENQUEUE ================= */

void hfsc_packet_in(struct rte_mbuf *m)
{
    hfsc_class_t *cl = hfsc_classify(m);
    if (!cl) return;

    if (!cl->active)
        hfsc_activate(cl, now_cycles());

    rte_ring_enqueue(cl->q, m);
}

/* ================= LINK SHARING ================= */

static hfsc_class_t *
hfsc_ls_select(hfsc_class_t *cl)
{
    if (cl->is_leaf)
        return cl;
```

```
    hfsc_class_t *best = NULL;
    for (int i = 0; i < cl->num_children; i++) {
        hfsc_class_t *c = cl->children[i];
        if (!c->active) continue;
        if (!best || c->virtual_time < best->virtual_time)
            best = c;
    }
    return best ? hfsc_ls_select(best) : NULL;
}

/* ================= SCHEDULER ================= */

static hfsc_class_t *
hfsc_select(uint64_t now)
{
    hfsc_class_t *rt = rt_heap_peek();
    if (rt && rt->eligible_time <= now)
        return rt;

    return hfsc_ls_select(root);
}

/* ================= ACCOUNTING ================= */

static void
```

```
hfsc_account(hfsc_class_t *cl, struct rte_mbuf *m, uint64_t now)
{
    uint32_t len = rte_pktmbuf_pkt_len(m);

    cl->virtual_time += len;
    cl->parent->virtual_time +=
        bytes_to_cycles(len, cl->parent->sc.m1);
    root->virtual_time +=
        bytes_to_cycles(len, root->sc.m1);

    if (cl->eligible_time <= now) {
        cl->deadline +=
            bytes_to_cycles(len, cl->sc.m1);
    }
}

/* ================= DEQUEUE ================= */

struct rte_mbuf *
hfsc_packet_out(void)
{
    uint64_t now = now_cycles();
    hfsc_class_t *cl = hfsc_select(now);
    if (!cl) return NULL;

    struct rte_mbuf *m;
```

```
    if (rte_ring_dequeue(cl->q, (void **)&m) < 0)
        return NULL;

    hfsc_account(cl, m, now);

    if (rte_ring_empty(cl->q)) {
        cl->active = false;
        rt_heap_remove(cl);
    }

    return m;
}

/* ================= INIT ================= */

static hfsc_class_t *
mk_class(hfsc_class_t *parent, uint64_t rate, bool leaf, const char *qname)
{
    hfsc_class_t *c = calloc(1, sizeof(*c));
    c->parent = parent;
    c->sc.m1 = rate;
    c->is_leaf = leaf;

    if (parent) {
        parent->children =
            realloc(parent->children,
```

```
                sizeof(void *) * (parent->num_children + 1));
        parent->children[parent->num_children++] = c;
    }

    if (leaf) {
        c->q = rte_ring_create(
            qname, QUEUE_SIZE,
            rte_socket_id(),
            RING_F_SP_ENQ | RING_F_SC_DEQ);
    }
    return c;
}

void hfsc_init(void)
{
    root  = mk_class(NULL, 100000000, false, NULL);
    site1 = mk_class(root, 50000000, false, NULL);
    site2 = mk_class(root, 50000000, false, NULL);

    site1_udp = mk_class(site1, 25000000, true, "s1_udp");
    site1_tcp = mk_class(site1, 25000000, true, "s1_tcp");

    site2_udp = mk_class(site2, 25000000, true, "s2_udp");
    site2_tcp = mk_class(site2, 25000000, true, "s2_tcp");
}
```