```
/******************************************************************
 * HFSC Scheduler — Hierarchical + RT heap + Link-sharing (paper-faithful)
 * DPDK dataplane implementation
 *
 * Implements:
 *  - Leaf deadlines (SCED-style) with eligible times (Eq. 7, 11)
 *  - Link-sharing via hierarchical virtual times (Eq. 12), SSF policy
 *  - Two min-heaps:
 *      * eligible_heap: among eligible leaves, pick min deadline
 *      * pending_heap: non-eligible leaves keyed by eligible_time
 *  - Piecewise-linear service curves (m1, m2, x) per paper §IV
 *
 * Notes:
 *  - Deadlines/eligible times are in wall-clock cycles (TSC)
 *  - Virtual times are in “service-normalized” cycles (via inverse curves)
 *  - Leaf classes guarantee service curves; interior classes use virtual times
 *  - Simplified piecewise-linear inverse computations for efficiency
 ******************************************************************/

#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>
#include <rte_byteorder.h>

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

#define QUEUE_SIZE     8192
#define MAX_CHILDREN   8
#define MAX_CLASSES    64
#define HEAP_CAPACITY  MAX_CLASSES

/* ================= SERVICE CURVE ================= */
/* Two-piece linear curve:
 *  - First segment slope m1 (bytes/sec) up to x bytes
 *  - Second segment slope m2 (bytes/sec) after x
 */
typedef struct {
    uint64_t m1;   /* bytes/sec */
    uint64_t m2;   /* bytes/sec */
    uint64_t x;    /* bytes */
} service_curve_t;

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

```
typedef struct hfsc_class {
    struct hfsc_class *parent;
    struct hfsc_class *children[MAX_CHILDREN];
    int num_children;

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

    /* Service curve */
    service_curve_t sc;

    /* Leaf RT state (wall-clock cycles) */
    uint64_t eligible_time; /* e_i */
    uint64_t deadline;      /* d_i */

    /* Leaf RT accounting (bytes) */
    uint64_t rt_service;    /* c_i: bytes served under RT criterion */

    /* Virtual time state (service-normalized cycles) */
    uint64_t vtime;         /* v_i */
    uint64_t total_service; /* w_i: bytes served total (RT + LS) */

    /* Virtual curve parameters (piecewise-linear inverse):
     * Vi(a_k; v) = min(prev, S_i(v - v_parent + w_i(a_k)))
     * We store slopes and offset to compute inverse quickly.
```

```
 */
  uint64_t vc_m1;  /* slope in bytes/sec for first segment (maps v->bytes) */
  uint64_t vc_m2;  /* slope in bytes/sec for second segment */
  uint64_t vc_x;   /* breakpoint in “virtual domain” (cycles) */
  uint64_t vc_off; /* offset (bytes) accumulated at activation */

  bool active;
} hfsc_class_t;

/* ================= HEAP (min-heap) ================= */

typedef struct {
  uint64_t key;
  hfsc_class_t *cl;
} heap_entry_t;

typedef struct {
  heap_entry_t *entries;
  int size;
  int capacity;
} min_heap_t;

static void heap_init(min_heap_t *h, int cap) {
  h->entries = (heap_entry_t *)calloc(cap, sizeof(heap_entry_t));
  h->size = 0;
  h->capacity = cap;
```

```
}

static void heap_push(min_heap_t *h, uint64_t key, hfsc_class_t *cl) {
    if (h->size >= h->capacity) return;
    int i = h->size++;
    h->entries[i].key = key;
    h->entries[i].cl = cl;
    while (i > 0) {
        int p = (i - 1) / 2;
        if (h->entries[p].key <= h->entries[i].key) break;
        heap_entry_t tmp = h->entries[p];
        h->entries[p] = h->entries[i];
        h->entries[i] = tmp;
        i = p;
    }
}

static bool heap_empty(min_heap_t *h) { return h->size == 0; }

static heap_entry_t heap_top(min_heap_t *h) { return h->entries[0]; }

static heap_entry_t heap_pop(min_heap_t *h) {
    heap_entry_t min = h->entries[0];
    h->entries[0] = h->entries[--h->size];
    int i = 0;
    while (1) {
```

```
        int l = 2 * i + 1, r = 2 * i + 2, s = i;
        if (l < h->size && h->entries[l].key < h->entries[s].key) s = l;
        if (r < h->size && h->entries[r].key < h->entries[s].key) s = r;
        if (s == i) break;
        heap_entry_t tmp = h->entries[s];
        h->entries[s] = h->entries[i];
        h->entries[i] = tmp;
        i = s;
    }
    return min;
}

/* ================= GLOBALS ================= */

static hfsc_class_t *root;
static hfsc_class_t *site1, *site2;
static hfsc_class_t *site1_udp, *site1_tcp, *site2_udp, *site2_tcp;

static min_heap_t eligible_heap; /* key = deadline */
static min_heap_t pending_heap;  /* key = eligible_time */

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

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_bytes_per_sec) {
```

```
    if (!rate_bytes_per_sec) return 0;
    return (bytes * rte_get_tsc_hz()) / rate_bytes_per_sec;
}

/* ================= SERVICE CURVE HELPERS ================= */
/* Deadline increment for a packet of length len under SCED:
 * If c_i + len <= x: use m1; else use m2.
 */
static inline uint64_t sc_deadline_delta(const service_curve_t *sc, uint64_t c_bytes, uint32_t len_bytes) {
    uint64_t before = c_bytes;
    uint64_t after  = c_bytes + len_bytes;
    if (after <= sc->x) {
        return bytes_to_cycles(len_bytes, sc->m1);
    } else if (before >= sc->x) {
        return bytes_to_cycles(len_bytes, sc->m2);
    } else {
        /* crosses breakpoint: split */
        uint64_t first = sc->x - before;
        uint64_t second = after - sc->x;
        return bytes_to_cycles(first, sc->m1) + bytes_to_cycles(second, sc->m2);
    }
}

/* Eligible curve E_i(a_k; t): for concave (m1 >= m2) equals D_i; for convex, cap by m2.
 * We implement the efficient rule from §IV:
```

```
 *  - concave: eligible == deadline
 *  - convex: eligible grows at m2 slope (i.e., cap RT allocation to asymptotic rate)
 */
static inline bool sc_is_concave(const service_curve_t *sc) { return sc->m1 >= sc->m2; }

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

static inline 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 || ip->version != 4) return NULL;

   uint32_t src = rte_be_to_cpu_32(ip->src_addr);
   uint32_t dst = rte_be_to_cpu_32(ip->dst_addr);
   /* 192.168.2.20 -> 192.168.2.30 */
   if (src != 0xC0A80214 || dst != 0xC0A8021E) 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;
   } else 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;
}

/* ================ VIRTUAL CURVE UPDATE (Eq. 12) ================= */
/* We maintain virtual curve inverse parameters:
 * Vi(a_k; v) = min(prev, S_i(v - v_parent(a_k)) + w_i(a_k))
 * For piecewise-linear S_i, we set:
 *  - vc_m1 = sc.m1, vc_m2 = sc.m2
 *  - vc_x  = v_break = cycles corresponding to x bytes at slope m1
 *  - vc_off = w_i(a_k) (bytes already served)
 * v_parent is folded into vtime initialization at activation.
 */
static void update_virtual_curve(hfsc_class_t *cl, uint64_t v_parent, uint64_t w_bytes) {
   cl->vc_m1 = cl->sc.m1;
   cl->vc_m2 = cl->sc.m2;
   cl->vc_x  = bytes_to_cycles(cl->sc.x, cl->sc.m1);
   cl->vc_off = w_bytes;
   /* Initialize v_i to max(v_i, v_parent) to keep siblings bounded (SSF with vs=(min+max)/2
approx) */
   if (cl->vtime < v_parent) cl->vtime = v_parent;
}

/* Advance virtual time by inverse of S_i for bytes 'len':
 * If v < vc_x: use m1; else use m2.
```

```
 */
static inline uint64_t vc_advance_cycles(hfsc_class_t *cl, uint32_t len_bytes) {
   /* Map bytes to virtual cycles using current segment slope.
    * We approximate by using m2 for fairness stability (bounded discrepancy),
    * but if cl->vtime < vc_x, we use m1 until crossing.
    */
   uint64_t v = cl->vtime;
   if (v < cl->vc_x) {
      /* first segment */
      uint64_t delta = bytes_to_cycles(len_bytes, cl->vc_m1);
      uint64_t newv = v + delta;
      if (newv <= cl->vc_x) return delta;
      /* crosses breakpoint: split */
      uint64_t first = cl->vc_x - v;
      uint64_t first_bytes = (first * cl->vc_m1) / rte_get_tsc_hz();
      uint64_t second_bytes = len_bytes - first_bytes;
      return first + bytes_to_cycles(second_bytes, cl->vc_m2);
   } else {
      return bytes_to_cycles(len_bytes, cl->vc_m2);
   }
}

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

static void set_active(hfsc_class_t *cl, uint64_t now) {
   if (cl->active) return;
```

```
    cl->active = true;

    if (cl->is_leaf) {
        /* Initialize RT state for new active period */
        cl->rt_service = 0;

        /* First packet's eligible/deadline:
         * Eligible: concave -> equals deadline; convex -> cap by m2 slope.
         * We start at now; next packet's increments computed on dequeue.
         */
        cl->eligible_time = now; /* eligible immediately for concave; for convex we gate via
pending_heap */
        cl->deadline = now;      /* will be advanced on dequeue by sc_deadline_delta */

        /* Insert into appropriate heap */
        if (sc_is_concave(&cl->sc)) {
            heap_push(&eligible_heap, cl->deadline, cl);
        } else {
            /* convex: not eligible until we allocate at m2 rate; start pending at now */
            heap_push(&pending_heap, cl->eligible_time, cl);
        }
    }

    /* Initialize virtual curve at activation */
    uint64_t v_parent = cl->parent ? cl->parent->vtime : 0;
    update_virtual_curve(cl, v_parent, cl->total_service);
```

```
    /* Bubble up activation */
    if (cl->parent) set_active(cl->parent, now);
}

static void set_passive(hfsc_class_t *cl) {
    if (!cl->active) return;
    cl->active = false;

    /* Bubble up: if parent has no active children, make it passive */
    hfsc_class_t *p = cl->parent;
    while (p) {
        bool any = false;
        for (int i = 0; i < p->num_children; i++) {
            if (p->children[i]->active) { any = true; break; }
        }
        if (any) break;
        p->active = false;
        p = p->parent;
    }
}

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

int hfsc_packet_in(struct rte_mbuf *m) {
    hfsc_class_t *cl = hfsc_classify(m);
```

```
    if (!cl || !cl->is_leaf) { rte_pktmbuf_free(m); return -1; }

    if (rte_ring_enqueue(cl->q, m) < 0) { rte_pktmbuf_free(m); return -1; }

    if (!cl->active) set_active(cl, now_cycles());
    return 0;
}

/* ================= RT HEAP MAINTENANCE ================= */

static void promote_pending(uint64_t now) {
    while (!heap_empty(&pending_heap) && heap_top(&pending_heap).key <= now) {
        heap_entry_t e = heap_pop(&pending_heap);
        if (e.cl->active) heap_push(&eligible_heap, e.cl->deadline, e.cl);
    }
}

/* ================= LINK-SHARING SELECTION (SSF) ================= */

static hfsc_class_t *ls_select(hfsc_class_t *cl) {
    if (!cl || !cl->active) return NULL;
    if (cl->is_leaf) return cl;

    hfsc_class_t *best = NULL;
    uint64_t best_v = UINT64_MAX;
    for (int i = 0; i < cl->num_children; i++) {
```

```
        hfsc_class_t *c = cl->children[i];
        if (!c->active) continue;
        if (c->vtime < best_v) { best_v = c->vtime; best = c; }
    }
    return ls_select(best);
}

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

static hfsc_class_t *hfsc_select(uint64_t now) {
    promote_pending(now);

    if (!heap_empty(&eligible_heap)) {
        heap_entry_t top = heap_top(&eligible_heap);
        if (top.cl->active && top.key <= now) return top.cl;
    }
    return ls_select(root);
}

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

static void account_packet(hfsc_class_t *leaf, struct rte_mbuf *m, uint64_t now, bool used_rt) {
    uint32_t len = rte_pktmbuf_pkt_len(m);

    /* Update leaf totals */
    leaf->total_service += len;
```

```
/* Virtual time advance at leaf */
leaf->vtime += vc_advance_cycles(leaf, len);

/* Propagate virtual time and totals up the tree */
hfsc_class_t *p = leaf->parent;
while (p) {
  p->total_service += len;
  /* SSF: keep vtime monotone and bounded among siblings */
  uint64_t dv = vc_advance_cycles(p, len);
  if (p->vtime < leaf->vtime) p->vtime = leaf->vtime; /* tighten to child min */
  p->vtime += dv;
  p = p->parent;
}

/* RT deadline/eligible updates if RT was used */
if (used_rt) {
  uint64_t delta = sc_deadline_delta(&leaf->sc, leaf->rt_service, len);
  leaf->rt_service += len;
  leaf->deadline += delta;

  if (sc_is_concave(&leaf->sc)) {
    /* eligible grows with deadline for concave */
    leaf->eligible_time = leaf->deadline;
  } else {
    /* convex: eligible grows at m2 slope only */
```

```c
            leaf->eligible_time += bytes_to_cycles(len, leaf->sc.m2);
        }
    }
}

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

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

    /* Pop from eligible heap if RT path */
    bool used_rt = false;
    if (!heap_empty(&eligible_heap) && heap_top(&eligible_heap).cl == cl &&
heap_top(&eligible_heap).key <= now) {
        (void)heap_pop(&eligible_heap);
        used_rt = true;
    }

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

    account_packet(cl, m, now, used_rt);

    /* If queue still has packets, update next head's deadline/eligible */
```

```
    if (!rte_ring_empty(cl->q)) {
        /* Peek next packet length: DPDK ring has no peek; we approximate by using typical MTU or defer until dequeue.
         * For correctness, we conservatively schedule next deadline on next dequeue (lazy update).
         */
        /* Reinsert into appropriate heap */
        if (sc_is_concave(&cl->sc)) {
            heap_push(&eligible_heap, cl->deadline, cl);
        } else {
            heap_push(&pending_heap, cl->eligible_time, cl);
        }
    } else {
        set_passive(cl);
    }

    return m;
}

/* ================= CLASS CREATION ================= */

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

```
    if (parent) 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;
}

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

void hfsc_init(void) {
    heap_init(&eligible_heap, HEAP_CAPACITY);
    heap_init(&pending_heap, HEAP_CAPACITY);

    /* Root: 100 Mbps */
    root  = mk_class(NULL, (service_curve_t){ .m1 = 12500000, .m2 = 12500000, .x = 0 }, false,
NULL);

    /* site1/site2: 50 Mbps each */
    site1 = mk_class(root, (service_curve_t){ .m1 = 6250000, .m2 = 6250000, .x = 0 }, false, NULL);
    site2 = mk_class(root, (service_curve_t){ .m1 = 6250000, .m2 = 6250000, .x = 0 }, false, NULL);

    /* site1 leaves: UDP 25 Mbps, TCP 25 Mbps (example split) */
    site1_udp = mk_class(site1, (service_curve_t){ .m1 = 25000000, .m2 = 25000000, .x = 0 },
true, "s1_udp");
```

```
    site1_tcp = mk_class(site1, (service_curve_t){ .m1 = 25000000, .m2 = 25000000, .x = 0 }, true,
"s1_tcp");

    /* site2 leaves: UDP 25 Mbps, TCP 25 Mbps */
    site2_udp = mk_class(site2, (service_curve_t){ .m1 = 25000000, .m2 = 25000000, .x = 0 },
true, "s2_udp");
    site2_tcp = mk_class(site2, (service_curve_t){ .m1 = 25000000, .m2 = 25000000, .x = 0 }, true,
"s2_tcp");

    /* Initialize virtual times to zero */
    root->vtime = site1->vtime = site2->vtime = 0;
    site1_udp->vtime = site1_tcp->vtime = site2_udp->vtime = site2_tcp->vtime = 0;
}

/* ================ OPTIONAL: EXPORTS ================ */

hfsc_class_t *hfsc_root(void) { return root; }
hfsc_class_t *hfsc_site1(void) { return site1; }
hfsc_class_t *hfsc_site2(void) { return site2; }
hfsc_class_t *hfsc_site1_udp(void) { return site1_udp; }
hfsc_class_t *hfsc_site1_tcp(void) { return site1_tcp; }
hfsc_class_t *hfsc_site2_udp(void) { return site2_udp; }
hfsc_class_t *hfsc_site2_tcp(void) { return site2_tcp; }


///*****************l2fwd integration***************************
```

```
hfsc_init();

while (!force_quit) {
   // RX burst
   struct rte_mbuf *pkts_burst[MAX_PKT_BURST];
   uint16_t nb_rx = rte_eth_rx_burst(portid, queueid, pkts_burst, MAX_PKT_BURST);

   for (int i = 0; i < nb_rx; i++) {
      hfsc_packet_in(pkts_burst[i]);
   }

   // Scheduler + TX
   struct rte_mbuf *m;
   int nb_tx = 0;
   struct rte_mbuf *tx_pkts[MAX_PKT_BURST];
   while ((m = hfsc_packet_out()) != NULL && nb_tx < MAX_PKT_BURST) {
      tx_pkts[nb_tx++] = m;
   }
   if (nb_tx > 0) {
      rte_eth_tx_burst(portid, queueid, tx_pkts, nb_tx);
   }
}
```