```
#ifndef HFSC_H

#define HFSC_H


#include <rte_mbuf.h>

#include <rte_ring.h>


#define QUEUE_SIZE 8192

#define MAX_CHILDREN 4

#define AVG_PKT_LEN 1500


typedef struct {

    uint64_t m1;    // bytes/sec - initial slope

    uint64_t d;     // us - delay for first segment (0 for linear)

    uint64_t m2;    // bytes/sec - asymptotic slope

} service_curve_t;


typedef struct {

    double x;       // start time (sec)

    double y;       // start bytes

    double sm1;     // slope 1 (bytes/sec)

    double sm2;     // slope 2

    double dx;      // x-length of first segment

    double dy;      // y-length of first segment

} runtime_sc_t;


typedef struct hfsc_class {
```

```
struct hfsc_class *parent;
struct hfsc_class *children[MAX_CHILDREN];
int num_children;
bool is_leaf;

struct rte_ring *q;           // leaf only

service_curve_t rsc;
service_curve_t fsc;
service_curve_t usc;

runtime_sc_t deadline;
runtime_sc_t eligible;
runtime_sc_t virtual;
runtime_sc_t ulimit;

uint64_t cumul;                // RT service (bytes)
uint64_t total;              // total service (bytes)

uint64_t cl_e;               // eligible time (cycles)
uint64_t cl_d;               // deadline (cycles)
uint64_t cl_vt;              // virtual time (cycles)
uint64_t cl_myf;               // my fit time (from USC)
uint64_t cl_f;              // final fit time = max(myf, cfmin)

uint32_t vtperiod;
```

```
    uint32_t parentperiod;


    bool active;
    uint64_t last_time;
} hfsc_class_t;


extern hfsc_class_t *root, *site1, *site2, *udp1, *tcp1, *udp2, *tcp2;


void hfsc_init(void);
int hfsc_packet_in(struct rte_mbuf *m);
struct rte_mbuf *hfsc_packet_out(void);


#endif /* HFSC_H */


//***********************************
#include "hfsc.h"
/****************************************************************
 * DPDK HFSC – Enhanced, Logically Faithful to Linux Kernel HFSC
 * Features: USC (upper limit), accurate vt period, better peek
 ****************************************************************/
#include <stdint.h>
#include <stdbool.h>
#include <stdlib.h>
#include <math.h>
#include <rte_mbuf.h>
#include <rte_ring.h>
```

```
#include <rte_cycles.h>
#include <rte_ip.h>
#include <rte_udp.h>
#include <rte_tcp.h>
#include <rte_ether.h>
#include <rte_byteorder.h>

/* ================= CONFIG ================= */
#define QUEUE_SIZE 8192
#define MAX_CHILDREN 4
#define AVG_PKT_LEN 1500  // fallback when peek fails

/* ================= SERVICE CURVE ================= */
typedef struct {
    uint64_t m1;    // bytes/sec - initial slope
    uint64_t d;     // us - delay for first segment (0 for linear)
    uint64_t m2;    // bytes/sec - asymptotic slope
} service_curve_t;

/* ================= RUNTIME SERVICE CURVE ================= */
typedef struct {
    double x;       // start time (sec)
    double y;       // start bytes
    double sm1;     // slope 1 (bytes/sec)
    double sm2;     // slope 2
    double dx;      // x-length of first segment
    double dy;      // y-length of first segment
} runtime_sc_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;             // leaf only

    service_curve_t rsc;            // real-time (deadline)
    service_curve_t fsc;            // fair/link-sharing
    service_curve_t usc;            // upper limit (optional)

    runtime_sc_t deadline;          // runtime D
    runtime_sc_t eligible;          // runtime E
    runtime_sc_t virtual;           // runtime V
    runtime_sc_t ulimit;            // runtime U (for USC)

    uint64_t cumul;                 // RT service (bytes)
    uint64_t total;                 // total service (bytes)

    uint64_t cl_e;                  // eligible time (cycles)
    uint64_t cl_d;                  // deadline (cycles)
    uint64_t cl_vt;              // virtual time (cycles)
    uint64_t cl_myf;             // my fit time (from USC)
    uint64_t cl_f;               // final fit time = max(myf, cfmin)

    uint32_t vtperiod;           // current virtual time period
    uint32_t parentperiod;       // parent's period when activated

    bool active;
    uint64_t last_time;          // last update/activation time
} hfsc_class_t;

/* ================= GLOBAL STATE ================= */
static hfsc_class_t *root;
static hfsc_class_t *site1, *site2;
static hfsc_class_t *udp1, *tcp1, *udp2, *tcp2;

/* ================= HELPERS ================= */
static inline uint64_t now_cycles(void) {
    return rte_get_tsc_cycles();
}

static inline double cycles_to_sec(uint64_t c) {
    return (double)c / rte_get_tsc_hz();
}

static inline double bytes_per_sec_to_per_cycle(uint64_t bps) {
    return (double)bps / rte_get_tsc_hz();
}

/* ================ RUNTIME CURVE MATH ================ */
static double rtsc_x2y(runtime_sc_t *rt, double x) {
    if (x <= rt->x) return rt->y;
    if (x <= rt->x + rt->dx)
        return rt->y + (x - rt->x) * rt->sm1;
    return rt->y + rt->dy + (x - rt->x - rt->dx) * rt->sm2;
}

static double rtsc_y2x(runtime_sc_t *rt, double y) {
    if (y <= rt->y) return rt->x;
    if (y <= rt->y + rt->dy)
        return rt->x + (y - rt->y) / rt->sm1;
    return rt->x + rt->dx + (y - rt->y - rt->dy) / rt->sm2;
}

static void rtsc_min(runtime_sc_t *rt, double new_x, double new_y,
                     double sm1, double sm2, double dx) {
    double y1 = rtsc_x2y(rt, new_x);
    double dy_new = dx * sm1;

    if (sm1 <= sm2) { // convex
        if (y1 < new_y) return;
        rt->x = new_x; rt->y = new_y; rt->dx = dx; rt->dy = dy_new;
        rt->sm1 = sm1; rt->sm2 = sm2;
        return;
    }

    // concave
    double y2 = rtsc_x2y(rt, new_x + dx);
    if (y2 <= new_y + dy_new) {
        rt->x = new_x; rt->y = new_y; rt->dx = dx; rt->dy = dy_new;
        rt->sm1 = sm1; rt->sm2 = sm2;
        return;
    }

    // intersect - approximate new dx
    double diff = y1 - new_y;
    double dsm = sm1 - sm2;
    double new_dx = (dsm > 0) ? diff / dsm : dx;
    double new_dy = new_dx * sm1;

    rt->x = new_x; rt->y = new_y; rt->dx = new_dx; rt->dy = new_dy;
    rt->sm1 = sm1; rt->sm2 = sm2;
}

/* ================ PEEK NEXT PACKET LENGTH ================ */
static uint32_t peek_next_len(struct rte_ring *ring) {
    if (rte_ring_empty(ring)) return AVG_PKT_LEN;
    void *obj;
    if (rte_ring_dequeue(ring, &obj) == 0) {
        uint32_t len = rte_pktmbuf_pkt_len((struct rte_mbuf *)obj);
        rte_ring_enqueue(ring, obj);  // put back
        return len;
    }
    return AVG_PKT_LEN;
}

/* ================= ACTIVATE / INIT CURVES ================= */
static void init_runtime_curve(runtime_sc_t *rt, double now_sec, double start_bytes,
                               uint64_t m1, uint64_t m2, uint64_t d) {
    double sm1 = bytes_per_sec_to_per_cycle(m1);
    double sm2 = bytes_per_sec_to_per_cycle(m2);
    double dx = (double)d / 1000000.0; // us → sec

    rt->x = now_sec;
    rt->y = start_bytes;
    rt->sm1 = sm1;
    rt->sm2 = sm2;
    rt->dx = dx;
    rt->dy = dx * sm1;
}

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

```
cl->active = true;
cl->last_time = now;

double now_sec = cycles_to_sec(now);

// Real-time curve
if (cl->rsc.m1 > 0 || cl->rsc.m2 > 0) {
    init_runtime_curve(&cl->deadline, now_sec, cl->cumul,
                       cl->rsc.m1, cl->rsc.m2, cl->rsc.d);
    cl->eligible = cl->deadline;

    // Convex → eligible becomes linear m2
    if (cl->rsc.m1 <= cl->rsc.m2) {
        cl->eligible.dx = 0;
        cl->eligible.dy = 0;
        cl->eligible.sm1 = bytes_per_sec_to_per_cycle(cl->rsc.m2);
        cl->eligible.sm2 = cl->eligible.sm1;
    }

    uint32_t next_len = peek_next_len(cl->q);
    cl->cl_e = (uint64_t)(rtsc_y2x(&cl->eligible, cl->cumul) * rte_get_tsc_hz());
    cl->cl_d = (uint64_t)(rtsc_y2x(&cl->deadline, cl->cumul + next_len) * rte_get_tsc_hz());
}

// Link-sharing curve
if (cl->fsc.m1 > 0 || cl->fsc.m2 > 0) {
        init_runtime_curve(&cl->virtual, now_sec, cl->total,
                           cl->fsc.m1, cl->fsc.m2, cl->fsc.d);
        cl->cl_vt = (uint64_t)(rtsc_y2x(&cl->virtual, cl->total) * rte_get_tsc_hz());
    }

    // Upper limit curve (USC)
    if (cl->usc.m1 > 0 || cl->usc.m2 > 0) {
        init_runtime_curve(&cl->ulimit, now_sec, cl->total,
                           cl->usc.m1, cl->usc.m2, cl->usc.d);
        cl->cl_myf = (uint64_t)(rtsc_y2x(&cl->ulimit, cl->total) * rte_get_tsc_hz());
    } else {
        cl->cl_myf = UINT64_MAX; // no limit
    }

    // VT period handling
    cl->vtperiod++;
    if (cl->parent) {
        cl->parentperiod = cl->parent->vtperiod;
    }

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

/* ================= 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->version != 4) return NULL;

    uint32_t src_ip = rte_be_to_cpu_32(ip->src_addr);
    uint32_t dst_ip = rte_be_to_cpu_32(ip->dst_addr);
    if (src_ip != 0xc0a80214 || dst_ip != 0xc0a8021e) return NULL; // 192.168.2.20 → .30

    uint8_t proto = ip->next_proto_id;
    uint16_t dst_port = 0;

    if (proto == IPPROTO_UDP) {
        struct rte_udp_hdr *udp = (struct rte_udp_hdr *)(ip + 1);
        dst_port = rte_be_to_cpu_16(udp->dst_port);
        if (dst_port == 5001) return udp1;
        if (dst_port == 6001) return udp2;
    } else if (proto == IPPROTO_TCP) {
        struct rte_tcp_hdr *tcp = (struct rte_tcp_hdr *)(ip + 1);
        dst_port = rte_be_to_cpu_16(tcp->dst_port);
        if (dst_port == 5002) return tcp1;
        if (dst_port == 6002) return tcp2;
    }
    return NULL;
}

/* ================ 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;
    }

    uint64_t now = now_cycles();
    if (!cl->active) hfsc_activate(cl, now);

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

    return 0;
}

/* ================= RT SELECT ================= */
static hfsc_class_t *hfsc_rt_select(uint64_t now) {
    hfsc_class_t *candidates[] = {udp1, tcp1, udp2, tcp2};
    hfsc_class_t *best = NULL;
    uint64_t min_d = UINT64_MAX;

    for (int i = 0; i < 4; i++) {
        hfsc_class_t *c = candidates[i];
        if (c->active && c->cl_e <= now && c->cl_d < min_d) {
            min_d = c->cl_d;
            best = c;
        }
    }
    return best;
}

/* ================= LS SELECT ================= */
static hfsc_class_t *hfsc_ls_select(hfsc_class_t *cl, uint64_t now) {
    if (!cl->active) return NULL;
    if (cl->is_leaf) return cl;

    // Find child with min cl_f (fit time)
    hfsc_class_t *best = NULL;
    uint64_t min_f = UINT64_MAX;

    for (int i = 0; i < cl->num_children; i++) {
        hfsc_class_t *child = cl->children[i];
        if (child->active && child->cl_f < min_f) {
            min_f = child->cl_f;
            best = child;
        }
    }

    if (!best) return NULL;
    return hfsc_ls_select(best, now);
}


/* ================= DEQUEUE & ACCOUNTING ================= */
struct rte_mbuf *hfsc_packet_out(void) {
    uint64_t now = now_cycles();
    if (!root->active) return NULL;


    hfsc_class_t *cl = hfsc_rt_select(now);
    bool is_realtime = (cl != NULL);


    if (!cl)
        cl = hfsc_ls_select(root, now);


    if (!cl || !cl->is_leaf) return NULL;


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


    uint32_t len = rte_pktmbuf_pkt_len(m);


    cl->total += len;
    if (is_realtime)
        cl->cumul += len;
```

```
double now_sec = cycles_to_sec(now);


// Update virtual curve
rtsc_min(&cl->virtual, now_sec, cl->total,
        bytes_per_sec_to_per_cycle(cl->fsc.m1),
        bytes_per_sec_to_per_cycle(cl->fsc.m2), 0);
cl->cl_vt = (uint64_t)(rtsc_y2x(&cl->virtual, cl->total) * rte_get_tsc_hz());


// Update USC (upper limit)
if (cl->usc.m1 > 0 || cl->usc.m2 > 0) {
    rtsc_min(&cl->ulimit, now_sec, cl->total,
            bytes_per_sec_to_per_cycle(cl->usc.m1),
            bytes_per_sec_to_per_cycle(cl->usc.m2), 0);
    cl->cl_myf = (uint64_t)(rtsc_y2x(&cl->ulimit, cl->total) * rte_get_tsc_hz());
}


// Update RT if applicable
if (cl->rsc.m1 > 0 || cl->rsc.m2 > 0) {
    uint32_t next_len = peek_next_len(cl->q);
    rtsc_min(&cl->deadline, now_sec, cl->cumul,
            bytes_per_sec_to_per_cycle(cl->rsc.m1),
            bytes_per_sec_to_per_cycle(cl->rsc.m2),
            (double)cl->rsc.d / 1000000.0);


    cl->eligible = cl->deadline;
    if (cl->rsc.m1 <= cl->rsc.m2) {
            cl->eligible.dx = 0;
            cl->eligible.dy = 0;
        }

        cl->cl_e = (uint64_t)(rtsc_y2x(&cl->eligible, cl->cumul) * rte_get_tsc_hz());
        cl->cl_d = (uint64_t)(rtsc_y2x(&cl->deadline, cl->cumul + next_len) * rte_get_tsc_hz());
    }

    if (rte_ring_empty(cl->q)) {
        cl->active = false;
        cl->vtperiod++;  // new period when reactivated
    }

    return m;
}

/* ================ INIT ================ */
void hfsc_init(void) {
    // Root - 100 Mbps (no USC)
    root = calloc(1, sizeof(*root));
    root->rsc = (service_curve_t){12500000, 0, 12500000};
    root->fsc = root->rsc;
    root->usc = (service_curve_t){0, 0, 0}; // no limit

    // site1 - 50 Mbps (with USC cap at 60 Mbps)
    site1 = calloc(1, sizeof(*site1));
    site1->parent = root;
    site1->rsc = (service_curve_t){6250000, 0, 6250000};
    site1->fsc = site1->rsc;
    site1->usc = (service_curve_t){7500000, 0, 7500000};
    root->children[root->num_children++] = site1;

    // udp1 - concave RT + USC
    udp1 = calloc(1, sizeof(*udp1));
    udp1->parent = site1;
    udp1->rsc = (service_curve_t){5000000, 10000, 1250000};
    udp1->fsc = (service_curve_t){1250000, 0, 1250000};
    udp1->usc = (service_curve_t){2000000, 0, 2000000};
    udp1->is_leaf = true;
    udp1->q = rte_ring_create("udp1_q", QUEUE_SIZE, rte_socket_id(), RING_F_SP_ENQ | RING_F_SC_DEQ);
    site1->children[site1->num_children++] = udp1;

    // tcp1 - linear
    tcp1 = calloc(1, sizeof(*tcp1));
    tcp1->parent = site1;
    tcp1->rsc = (service_curve_t){5000000, 0, 5000000};
    tcp1->fsc = tcp1->rsc;
    tcp1->usc = (service_curve_t){6000000, 0, 6000000};
    tcp1->is_leaf = true;
    tcp1->q = rte_ring_create("tcp1_q", QUEUE_SIZE, rte_socket_id(), RING_F_SP_ENQ | RING_F_SC_DEQ);
    site1->children[site1->num_children++] = tcp1;
    // site2 - 50 Mbps (with USC cap at 60 Mbps)
    site2 = calloc(1, sizeof(*site2));
    site2->parent = root;
    site2->rsc = (service_curve_t){6250000, 0, 6250000};
    site2->fsc = site2->rsc;
    site2->usc = (service_curve_t){7500000, 0, 7500000};
    root->children[root->num_children++] = site2;

    // udp2 - concave RT + USC
    udp2 = calloc(1, sizeof(*udp2));
    udp2->parent = site2;
    udp2->rsc = (service_curve_t){5000000, 10000, 1250000};
    udp2->fsc = (service_curve_t){1250000, 0, 1250000};
    udp2->usc = (service_curve_t){2000000, 0, 2000000};
    udp2->is_leaf = true;
    udp2->q = rte_ring_create("udp2_q", QUEUE_SIZE, rte_socket_id(), RING_F_SP_ENQ |
RING_F_SC_DEQ);
    site2->children[site2->num_children++] = udp2;

    // tcp2 - linear
    tcp2 = calloc(1, sizeof(*tcp2));
    tcp2->parent = site2;
    tcp2->rsc = (service_curve_t){5000000, 0, 5000000};
    tcp2->fsc = tcp2->rsc;
    tcp2->usc = (service_curve_t){6000000, 0, 6000000};
    tcp2->is_leaf = true;

    tcp2->q = rte_ring_create("tcp2_q", QUEUE_SIZE, rte_socket_id(), RING_F_SP_ENQ |
RING_F_SC_DEQ);

    site2->children[site2->num_children++] = tcp2;

}


//***************************************

#include <stdio.h>

#include <stdint.h>

#include <signal.h>

#include <rte_eal.h>

#include <rte_ethdev.h>

#include <rte_mbuf.h>

#include <rte_mempool.h>

#include <rte_cycles.h>

#include "hfsc.h"


#define RX_RING_SIZE      1024

#define TX_RING_SIZE      1024

#define NUM_MBUFS         8191

#define MBUF_CACHE_SIZE   250

#define BURST_SIZE        32


static volatile bool force_quit = false;


static void signal_handler(int signum) {
    if (signum == SIGINT || signum == SIGTERM) {
        printf("\nCaught signal %d, shutting down...\n", signum);
        force_quit = true;
    }
}

int main(int argc, char **argv) {
    struct rte_mempool *mbuf_pool;
    int ret;

    // Initialize DPDK EAL
    ret = rte_eal_init(argc, argv);
    if (ret < 0)
        rte_exit(EXIT_FAILURE, "EAL init failed\n");

    signal(SIGINT, signal_handler);
    signal(SIGTERM, signal_handler);

    // Check ports (need at least 2)
    uint16_t nb_ports = rte_eth_dev_count_avail();
    if (nb_ports < 2)
        rte_exit(EXIT_FAILURE, "Need at least 2 ports\n");

    // Create mbuf pool
    mbuf_pool = rte_pktmbuf_pool_create("MBUF_POOL", NUM_MBUFS * nb_ports,
                                        MBUF_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE,
                                        rte_socket_id());
    if (!mbuf_pool)
        rte_exit(EXIT_FAILURE, "Cannot create mbuf pool\n");

    // Configure & start ports (port 0 = RX, port 1 = TX)
    struct rte_eth_conf port_conf = {0};
    for (uint16_t port = 0; port < 2; port++) {
        ret = rte_eth_dev_configure(port, 1, 1, &port_conf);
        if (ret < 0)
            rte_exit(EXIT_FAILURE, "Cannot configure port %u\n", port);

        ret = rte_eth_rx_queue_setup(port, 0, RX_RING_SIZE,
                                     rte_eth_dev_socket_id(port), NULL, mbuf_pool);
        if (ret < 0)
            rte_exit(EXIT_FAILURE, "rx queue setup failed on port %u\n", port);

        ret = rte_eth_tx_queue_setup(port, 0, TX_RING_SIZE,
                                     rte_eth_dev_socket_id(port), NULL);
        if (ret < 0)
            rte_exit(EXIT_FAILURE, "tx queue setup failed on port %u\n", port);

        rte_eth_dev_start(port);
        rte_eth_promiscuous_enable(port);
    }

    // Initialize HFSC scheduler
```

```
hfsc_init();
printf("HFSC scheduler initialized.\n");
printf("Forwarding: RX port 0 → HFSC → TX port 1\n");

// Main poll loop
while (!force_quit) {
    // RX from port 0
    struct rte_mbuf *rx_pkts[BURST_SIZE];
    uint16_t nb_rx = rte_eth_rx_burst(0, 0, rx_pkts, BURST_SIZE);

    for (uint16_t i = 0; i < nb_rx; i++) {
        if (hfsc_packet_in(rx_pkts[i]) != 0) {
            rte_pktmbuf_free(rx_pkts[i]);
        }
    }

    // Dequeue from HFSC & TX to port 1
    struct rte_mbuf *tx_pkts[BURST_SIZE];
    uint16_t nb_tx = 0;

    while (nb_tx < BURST_SIZE) {
        struct rte_mbuf *m = hfsc_packet_out();
        if (!m) break;
        tx_pkts[nb_tx++] = m;
    }
```

```
        if (nb_tx > 0) {
            uint16_t nb_sent = rte_eth_tx_burst(1, 0, tx_pkts, nb_tx);
            for (uint16_t i = nb_sent; i < nb_tx; i++) {
                rte_pktmbuf_free(tx_pkts[i]);
            }
        }
    }

    // Cleanup
    printf("Shutting down...\n");
    for (uint16_t port = 0; port < 2; port++) {
        rte_eth_dev_stop(port);
    }

    rte_eal_cleanup();
    return 0;
}

//************************************************
# SPDX-License-Identifier: BSD-3-Clause
# Copyright(c) 2026 Manikant

include $(RTE_SDK)/mk/rte.vars.mk

APP = dpdk-hfsc_forward
```

```
SRCS-y += main.c

SRCS-y += hfsc.c


CFLAGS += -O3 -g -I.


LIBS += -lrte_eal -lrte_mbuf -lrte_mempool_ring -lrte_ring -lrte_ethdev

LIBS += -lrte_bus_pci -lrte_pci -lrte_kvargs -ldl -lpthread -lnuma


include $(RTE_SDK)/mk/rte.extapp.mk
```

```
/*******************************
# Build (meson/ninja - modern DPDK style)

meson setup build

ninja -C build


# Or classic make (if your DPDK version uses it)

make -C examples/hfsc_forward T=x86_64-native-linux-gcc


# Run (very important: use --file-prefix to avoid conflicts)

sudo ./build/examples/dpdk-hfsc_forward \

   -l 0-3 \

   -n 4 \

   --huge-dir=/mnt/huge \

   --file-prefix=hfsc \

   --proc-type=primary
```

```
ninja -C build   # or make as above
sudo ./build/examples/dpdk-hfsc_forward \
    -l 0-3 -n 4 --huge-dir=/mnt/huge --file-prefix=hfsc
```