

```
*****  
* DPDK HFSC Scheduler – Enhanced, Single-File Version  
* Faithful to HFSC paper (Stoica et al.) + Linux kernel logic  
* Features: USC enforcement, peek next length, vtperiod, runtime curves  
*****/
```

```
#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>  
  
#include <math.h>  
  
#include <stdio.h>  
  
#include <stdint.h>  
  
#include <stdbool.h>  
  
#include <stdlib.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_vtadj;              // VT adjustment for fairness
uint64_t cl_myf;                // my fit time (from USC)
uint64_t cl_f;                // final fit time = max(myf, min_sibling_f)

/* ADDED: Track active children */
struct hfsc_class *active_children[MAX_CHILDREN];
int num_active_children;

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;

}

```

```

/* ====== 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;

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

```

```

/* ====== COMPUTE CL_F FUNCTION ====== */

static void compute_cl_f(hfsc_class_t *cl) {
    if (cl->parent == NULL) {
        /* Root class: cl_f = cl_myf */
        cl->cl_f = cl->cl_myf;
        return;
    }

```

```

/* Find minimum cl_f among active siblings */
uint64_t min_sibling_f = UINT64_MAX;
```

```

bool found_sibling = false;

for (int i = 0; i < cl->parent->num_children; i++) {
    hfsc_class_t *sibling = cl->parent->children[i];
    if (sibling != cl && sibling->active) {
        if (sibling->cl_f < min_sibling_f) {
            min_sibling_f = sibling->cl_f;
            found_sibling = true;
        }
    }
}

if (!found_sibling) {
    /* No active siblings */
    cl->cl_f = cl->cl_myf;
} else {
    /* cl_f = max(cl_myf, min_sibling_f) */
    cl->cl_f = (cl->cl_myf > min_sibling_f) ? cl->cl_myf : min_sibling_f;
}
}

/* ===== DEACTIVATION FUNCTION ===== */
static void hfsc_deactivate(hfsc_class_t *cl) {
    if (!cl->active) return;

    /* Remove from parent's active children list */
}

```

```

if (cl->parent) {

    for (int i = 0; i < cl->parent->num_active_children; i++) {

        if (cl->parent->active_children[i] == cl) {

            /* Shift remaining elements */

            for (int j = i; j < cl->parent->num_active_children - 1; j++) {

                cl->parent->active_children[j] = cl->parent->active_children[j + 1];

            }

            cl->parent->num_active_children--;

            break;

        }

    }

}

/* Recompute cl_f for siblings */

for (int i = 0; i < cl->parent->num_children; i++) {

    hfsc_class_t *sibling = cl->parent->children[i];

    if (sibling->active) {

        compute_cl_f(sibling);

    }

}

}

cl->active = false;

cl->num_active_children = 0; /* Clear active children list */

}

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

```

```

static void hfsc_activate(hfsc_class_t *cl, uint64_t now) {

    if (cl->active) return;

    cl->active = true;

    cl->last_time = now;

    /* Initialize active children tracking */

    cl->num_active_children = 0;

    if (cl->parent) {

        cl->parent->active_children[cl->parent->num_active_children++] = cl;

    }

    double now_sec = cycles_to_sec(now);

    // Real-time

    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;

        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

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

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;
}

/* Compute cl_f */

compute_cl_f(cl);

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;

    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 – WITH USC ENFORCEMENT ===== */

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 &&          // eligible time

        c->cl_myf <= now &&         // USC fit time – enforced

        c->cl_d < min_d) {

        min_d = c->cl_d;

        best = c;
    }
}

return best;
}

/* ===== LS SELECT – WITH USC ENFORCEMENT ===== */

static hfsc_class_t *hfsc_ls_select(hfsc_class_t *cl, uint64_t now) {

    if (!cl->active) return NULL;

    // Enforce this class's own USC

    if (cl->cl_myf > now) return NULL;

    if (cl->is_leaf) return cl;
}

```

```

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_myf <= now && // child must be fit (USC)
        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);

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());

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());
}

```

```

}

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());
}

/* Recompute cl_f after service */
compute_cl_f(cl);

/* Recompute cl_f for siblings */
if (cl->parent) {
    for (int i = 0; i < cl->parent->num_children; i++) {
        hfsc_class_t *sibling = cl->parent->children[i];
    }
}

```

```

        if (sibling->active && sibling != cl) {
            compute_cl_f(sibling);
        }
    }

}

if (rte_ring_empty(cl->q)) {
    hfsc_deactivate(cl);
    cl->vtperiod++;
}

return m;
}

/* ===== INIT ===== */
void hfsc_init(void) {
    root = calloc(1, sizeof(*root));
    root->rsc = (service_curve_t){12500000, 0, 12500000};
    root->fsc = root->rsc;
    root->usc = (service_curve_t){12500000, 0, 12500000}; // Root capped at 100 Mbps
    root->num_active_children = 0;
    root->cl_myf = UINT64_MAX;
    root->cl_f = root->cl_myf;

    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};

site1->num_active_children = 0;

site1->cl_myf = UINT64_MAX;

site1->cl_f = site1->cl_myf;

root->children[root->num_children++] = site1;

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->num_active_children = 0;

udp1->cl_myf = UINT64_MAX;

udp1->cl_f = udp1->cl_myf;

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 = 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->num_active_children = 0;  
tcp1->cl_myf = UINT64_MAX;  
tcp1->cl_f = tcp1->cl_myf;  
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 = 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};  
site2->num_active_children = 0;  
site2->cl_myf = UINT64_MAX;  
site2->cl_f = site2->cl_myf;  
root->children[root->num_children++] = site2;  
  
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->num_active_children = 0;  
udp2->cl_myf = UINT64_MAX;
```

```
    udp2->cl_f = udp2->cl_myf;

    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 = 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->num_active_children = 0;

tcp2->cl_myf = UINT64_MAX;

tcp2->cl_f = tcp2->cl_myf;

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;

}
```