DPM implementation of Open Event Machine

By

Shyam Pujar

10/09/2018

Agenda

- Typical Application view
- Open event machine adaptation view
- Typical Linux system View
- Typical event machine system view
- Application scheduling in Linux
- Application queue scheduling in Open Event Machine
- DPM implementation of event machine
- DPM event machine use in ASBTS
- DPM test applications

1. PID allocated for a process by kernel

```
int main(int argc, char* argv[])
{
   initialization(); /** Application specific initialization*/
   interface_creation(): /** Interface creation refers to opening interface/channels
                            towards the peers. peers could be another application within
                            same machine/ outside the machine */

   infinite_loop()
   {
        read_messages_from_interface();
        switch(message_type)
        {
           msg1:  msg1_handling();
           msg2: msg2_handling();
           .
           .
           .
           .
           msgN: msgN_handling();
           default: drop_message();
        }

   }

   delete_interfaces();
   release_resources_created_during_init();
}
```

2. Interface/channel creation
Could be
1. Socket
2. Pipes
3. FIFO
4. Message Queue
5. 3rd provided interface, which holds N num of messages/events

Kernel takes care of scheduling/descheduling process.
Application reads/write from/to interface when scheduled

2.1 PID to interface mapping is taken care by Kernel. Application uses the id given by kernel to read/write Messages to interface.

Mapping/adaption of the typical application to open event machine model

Absent as event driven model.

Event machine provides event, event_type, Queue_id , queue_context to receive_func.

em_queue_t
em_queue_create(const char *name, em_queue_type_t type, em_queue_prio_t prio, em_queue_group_t queue_group)

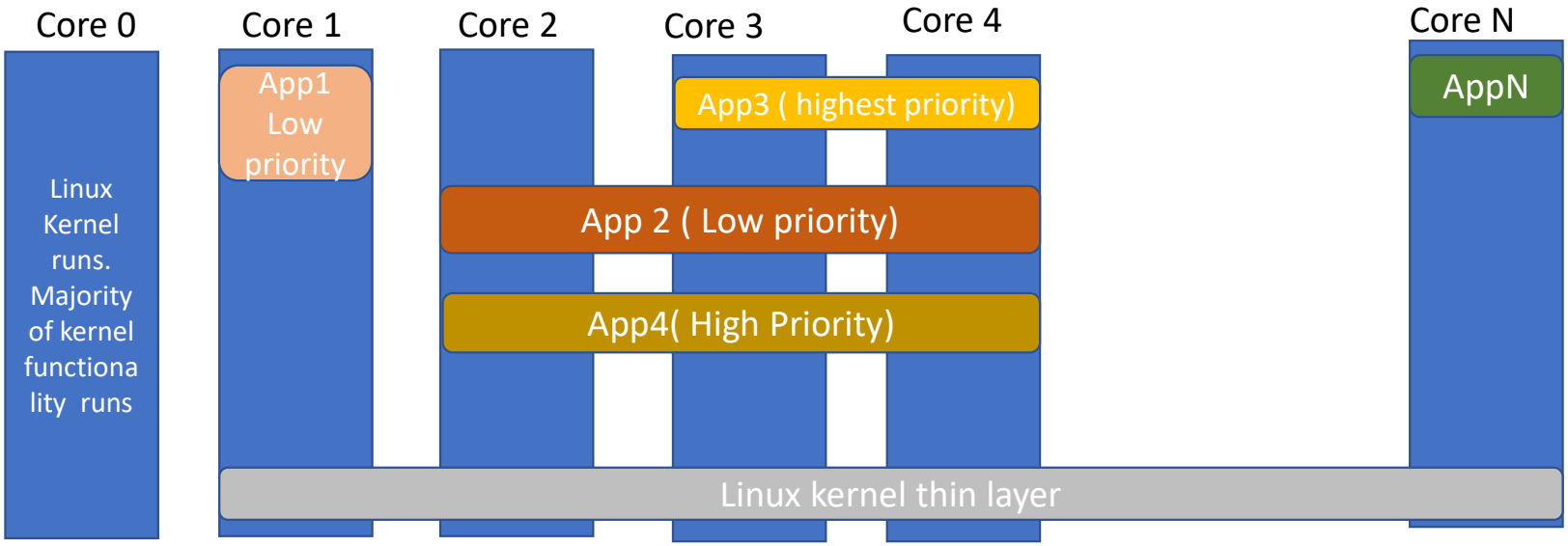
Application events/message holder

```
int main(int argc, char* argv[])
{
   initialization(); /** Application specific initialization*/
   interface_creation(); /** Interface creation refers to opening interface/channels
                            towards the peers. peers could be another application within
                            same machine/ outside the machine */

   infinite_loop()
   {
        read_messages_from_interface();
        switch(message_type)
        {
           msg1:  msg1_handling();
           msg2: msg2_handling();
           .
           .
           .
           .
           msgN: msgN_handling();
           default: drop_message();
        }

   }

   delete_interfaces();
   release_resources_created_during_init();
}
```

Em_eo_t em_eo_create(
Eo_Start_func,
Eo_local_start_func,
Em_Stop_func
Em_stop_local_func
receive_func,
Eo_context
)

em_status_t
queue_delete(queue_elem_t *const queue_elem)

Typical System View in Linux



Logic of setting affinity coremask;

Bit set corresponds to core on which application can run/be scheduled by kernel

App1 coremask: 0x2
App2 coremask: 0x1C
App3 coremask: 0x18

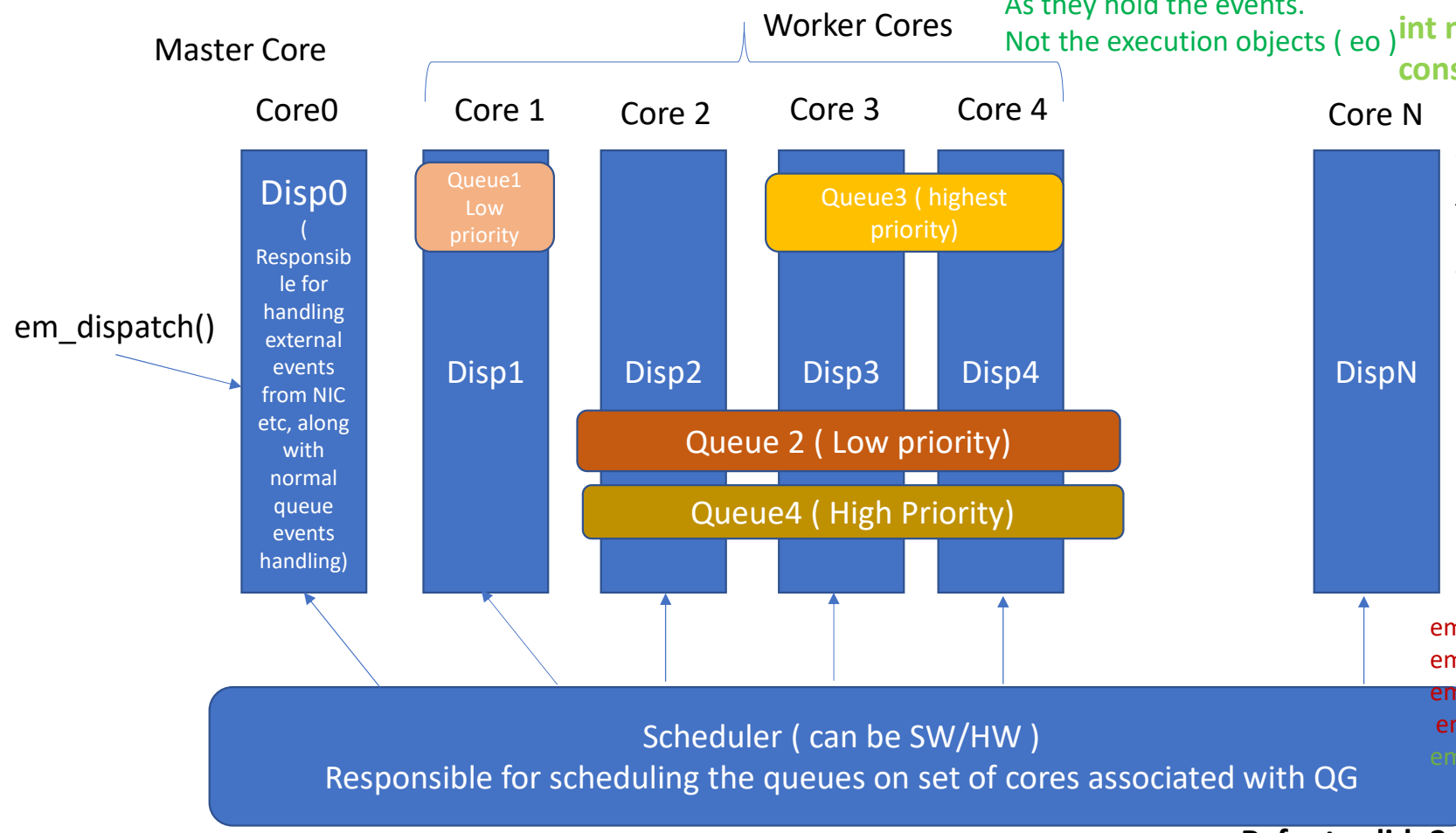
Kernel guarantess scheduling of processes only on Those cores, for which affinity is set.
Highest priority process , among those, which Can be scheduled on the free core, gets maximum Chance for core utilization

App1 → pinned to run on core1 only
App2, 4 → pinned to run on core2,3,4
App3 → pinned to run on core 3, 4
App N → pinned to run on core N
Pinning is done by setting affinity using taskset command/ sched_affinity API

Typical Event machine system view

em_queue_group_t
em_queue_group_create(
const char *name,
const em_core_mask_t *mask,
int num_notif,
const em_notif_t *notif_tbl)

Queues are schedulable
As they hold the events.
Not the execution objects (eo)



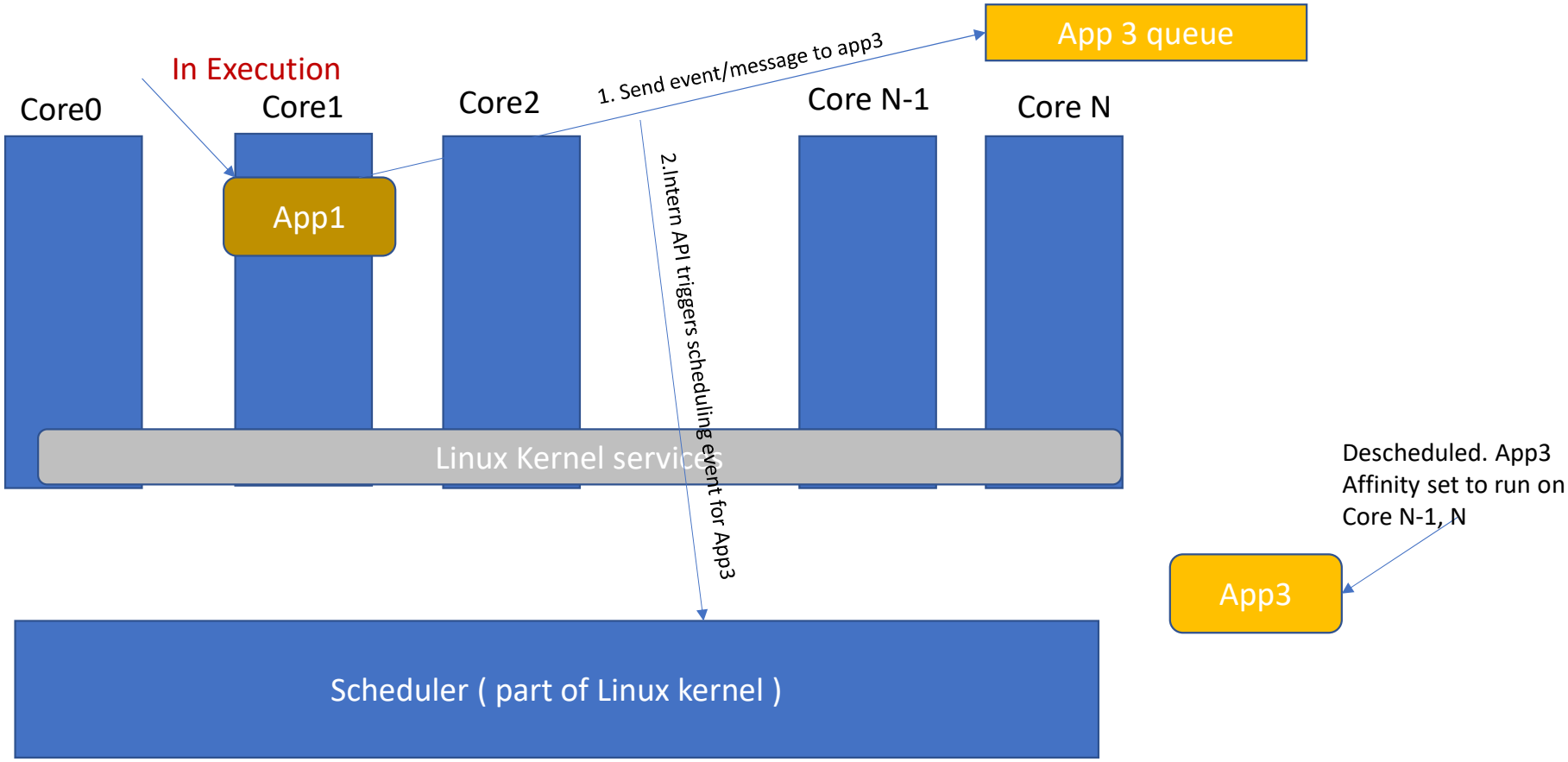
Configuring queues to run Using Queue Group(QG) Concept similar To setting affinity to a process in linux. QG need to be created in System before queue creation in system.

QG1= coremask 0x2
QG2= coremask 0x1E
QG3=coremask 0x18
QG4= coremask 0x1E

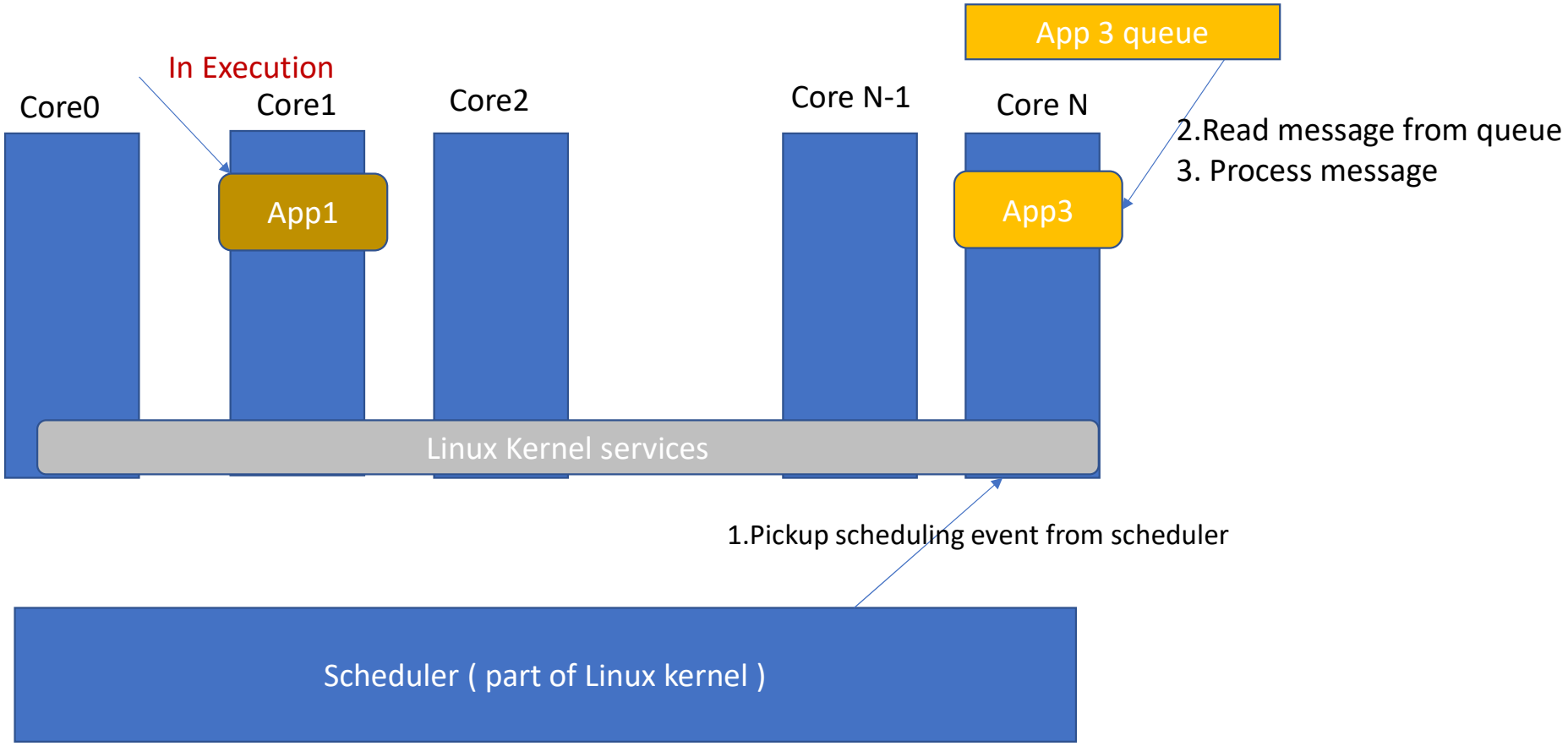
em_queue_t
em_queue_create(const char *name,
em_queue_type_t type,
em_queue_prio_t prio,
em_queue_group_t queue_group)

Refer to slide2 , for understanding queue

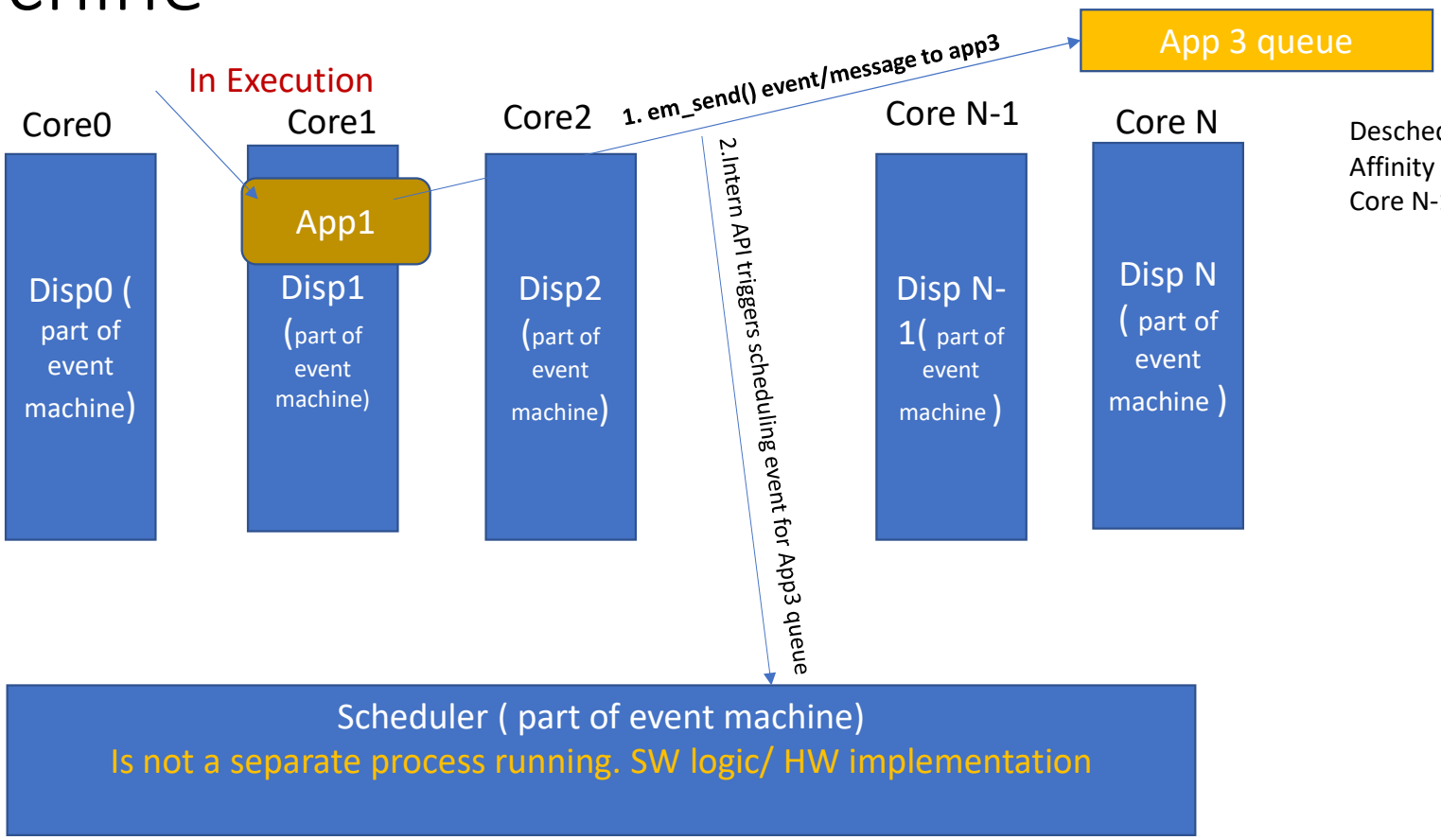
Application Scheduling in linux



Application Scheduling in linux

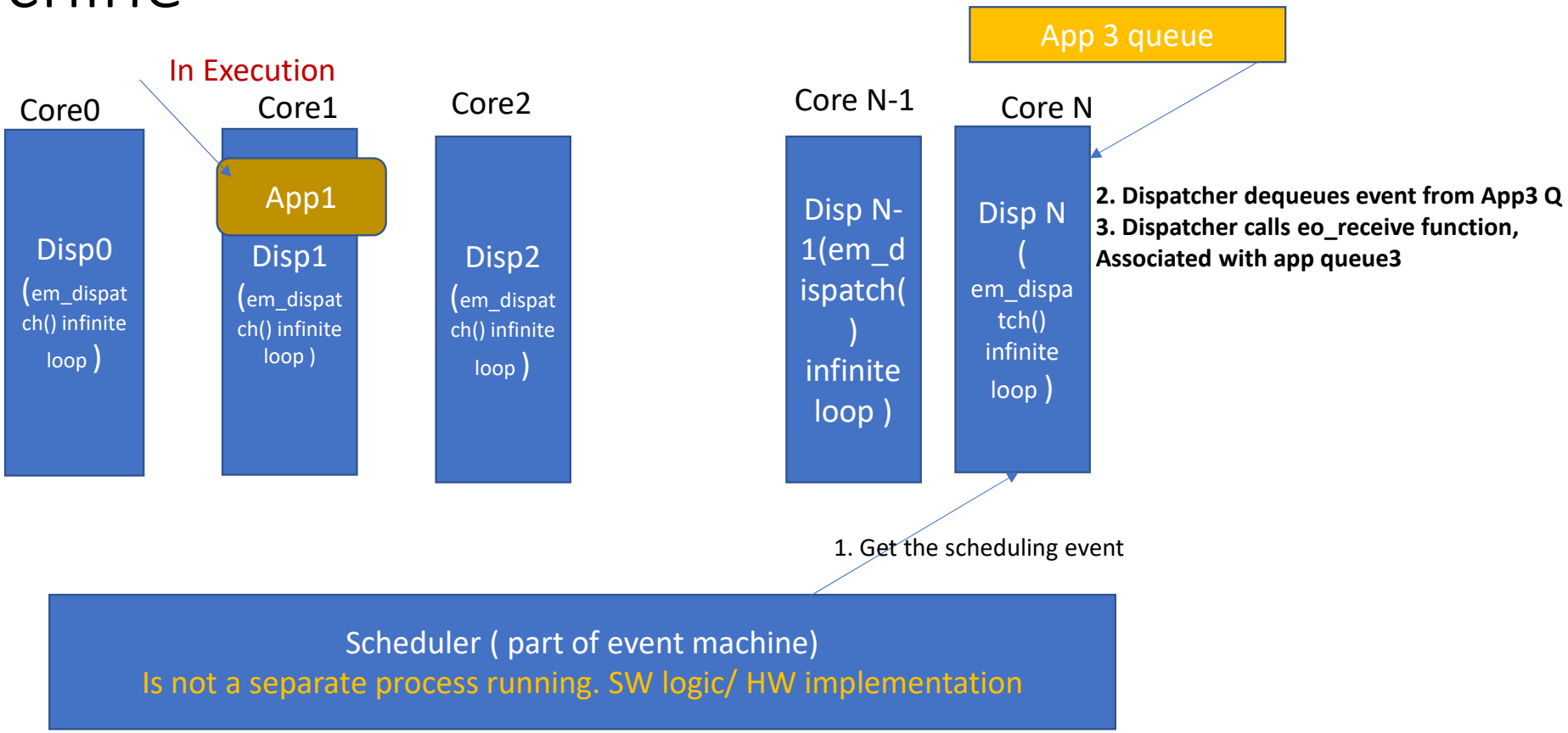


Application Queue Scheduling in event machine



Descheduled. App queue 3
Affinity set to run on
Core N-1, N (**recall Queue Group**)

Application Queue Scheduling in event machine

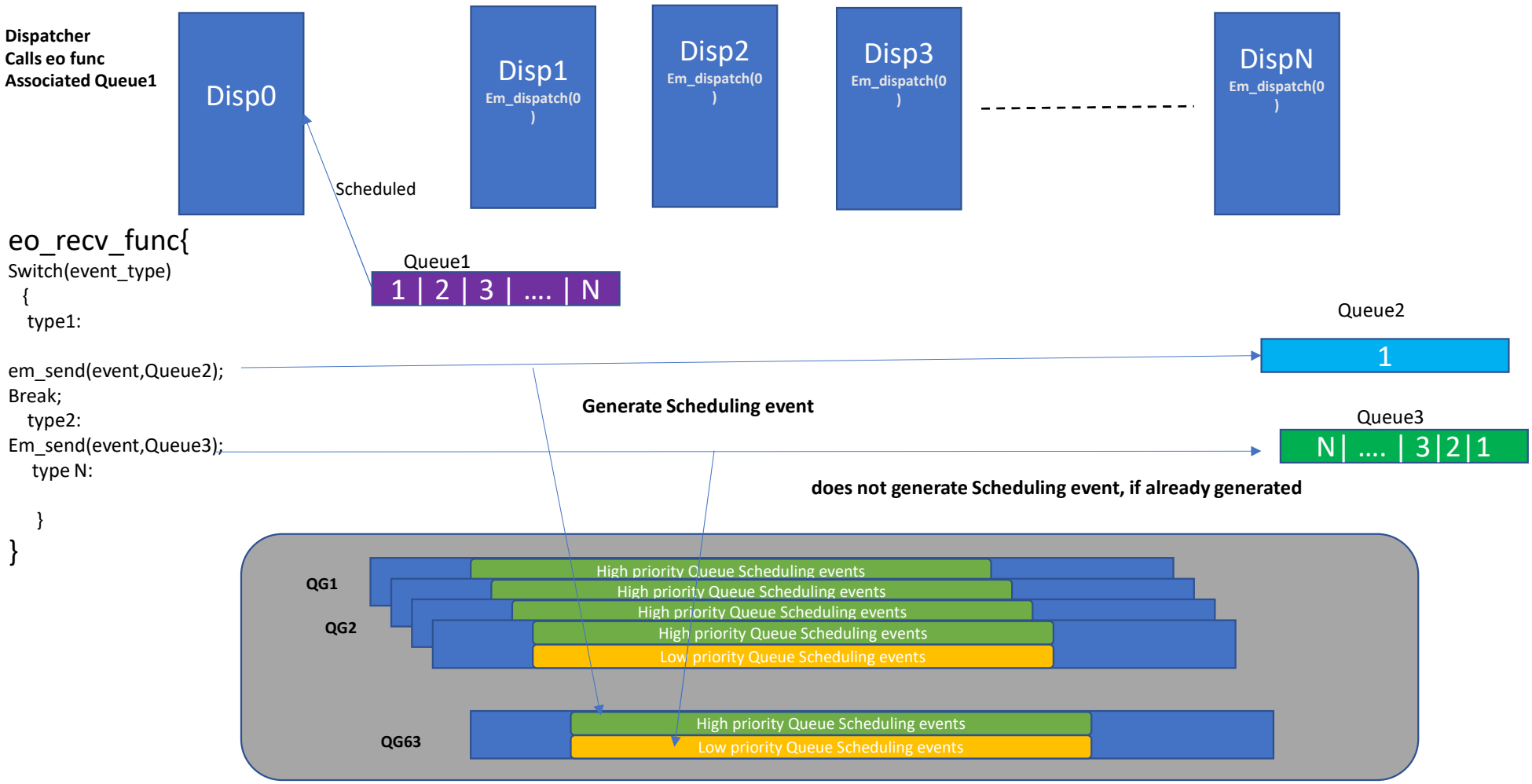


Procedure to create and start application in event machine

- Queue Group Create
- Eo create
- Queue create
- Attach queue to EO
- Enable the Queue
- Start EO

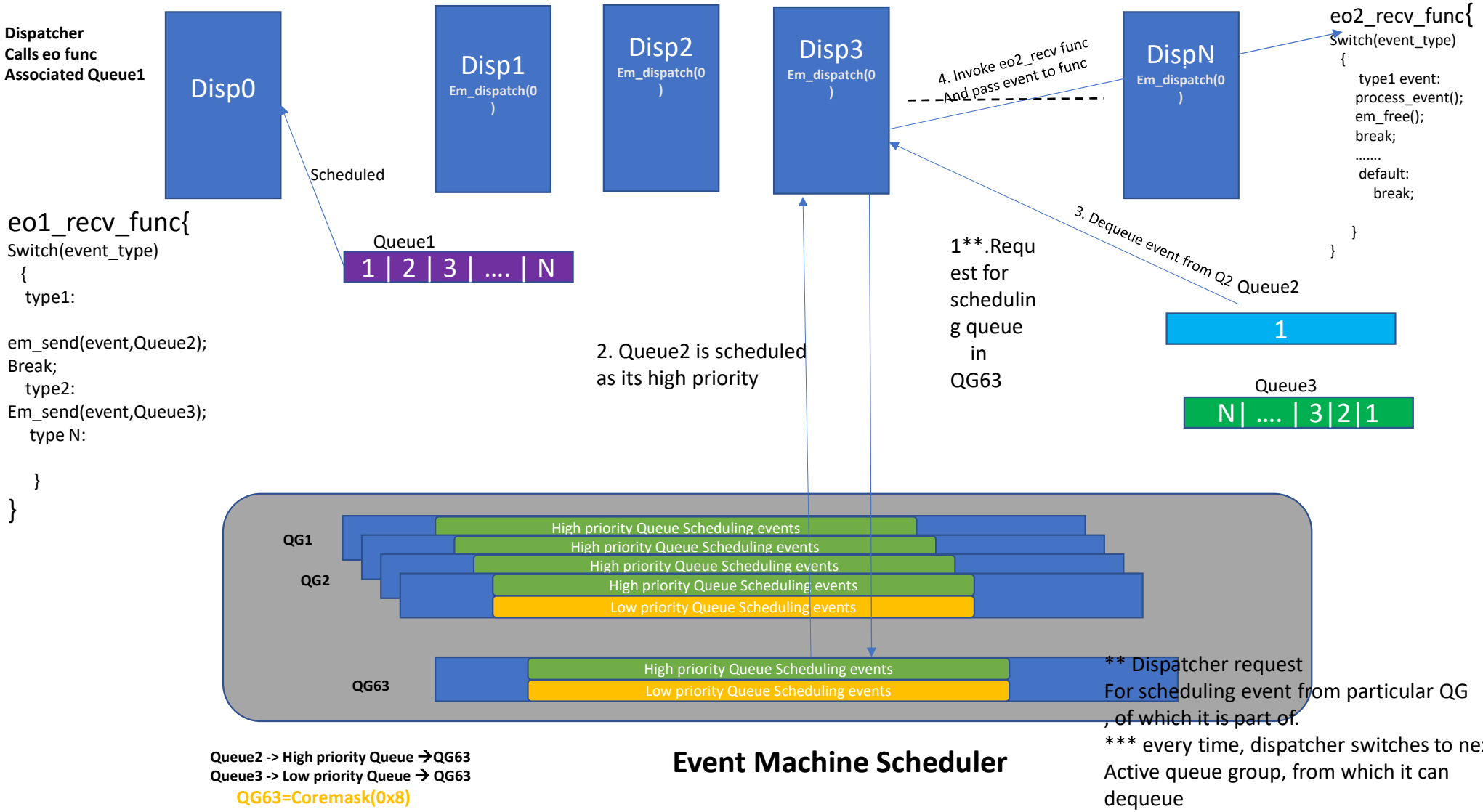
To be done for every application
In a product

DPM implementation of event machine



Queue2 -> High priority Queue →QG63
Queue3 -> Low priority Queue → QG63
QG63=Coremask(0x8)

Event Machine Scheduler

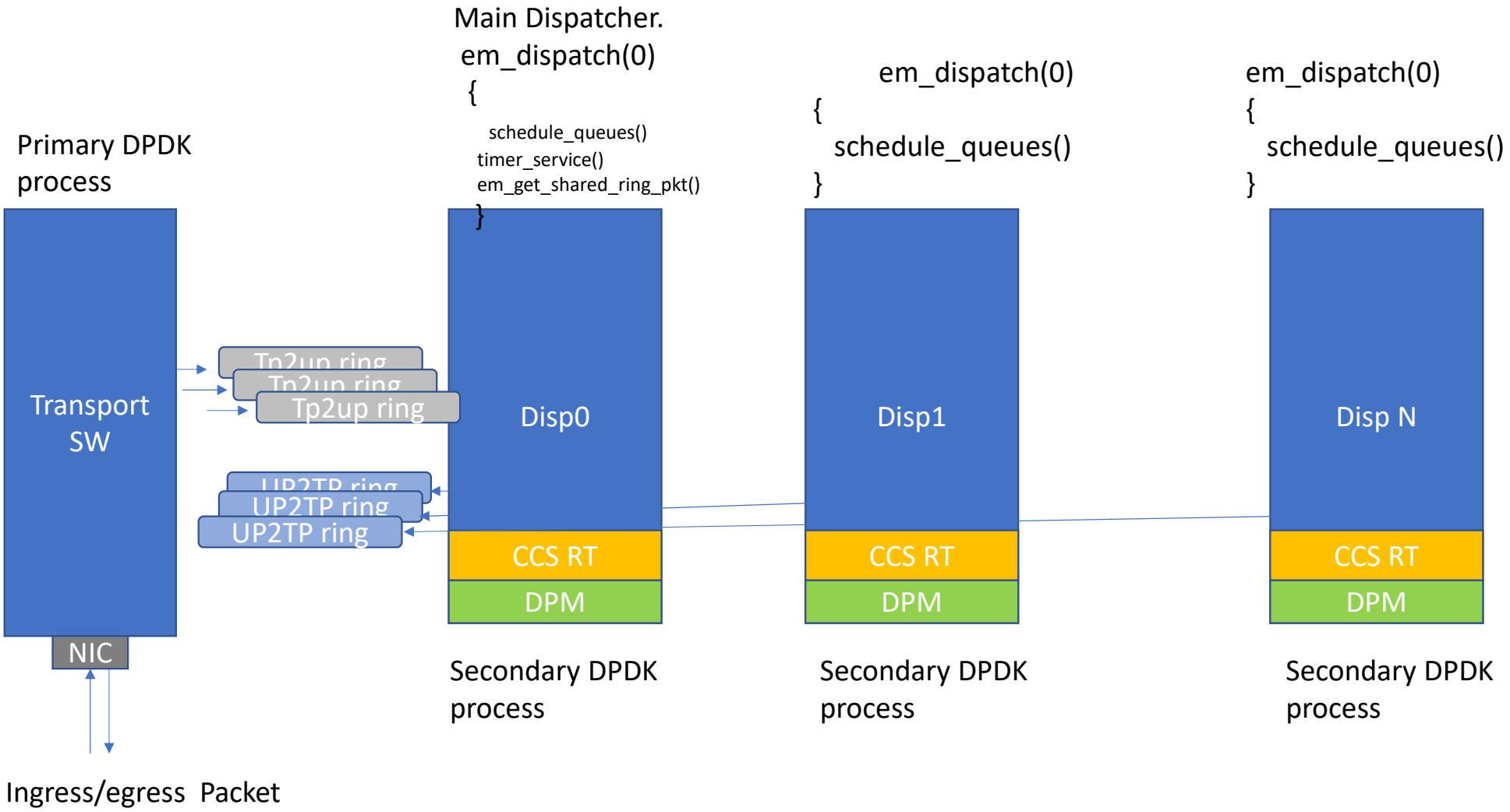

Dispatcher
Calls eo func
Associated Queue1
Disp0
Disp1
Em_dispatch(0
)
Disp2
Em_dispatch(0
)
Disp3
Em_dispatch(0
)
DispN
Em_dispatch(0
)
Scheduled
4. Invoke eo2_recv func
And pass event to func
3. Dequeue event from Q2
eo2_recv_func{
Switch(event_type)
{
type1 event:
process_event();
em_free();
break;
.....
default:
break;
}
}
eo1_recv_func{
Switch(event_type)
{
type1:
em_send(event,Queue2);
Break;
type2:
Em_send(event,Queue3);
type N:
}
}
Queue1
1 | 2 | 3 | .... | N
1**.Request for scheduling queue in QG63
2. Queue2 is scheduled as its high priority
Queue2
1
Queue3
N| .... | 3|2|1
QG1
QG2
QG63
High priority Queue Scheduling events
High priority Queue Scheduling events
High priority Queue Scheduling events
High priority Queue Scheduling events
Low priority Queue Scheduling events
High priority Queue Scheduling events
Low priority Queue Scheduling events
** Dispatcher request
For scheduling event from particular QG
, of which it is part of.
*** every time, dispatcher switches to next
Active queue group, from which it can
dequeue
Scheduling events
Queue2 -> High priority Queue →QG63
Queue3 -> Low priority Queue → QG63
QG63=Coremask(0x8)
Event Machine Scheduler

Dequeuing of events by dispatcher

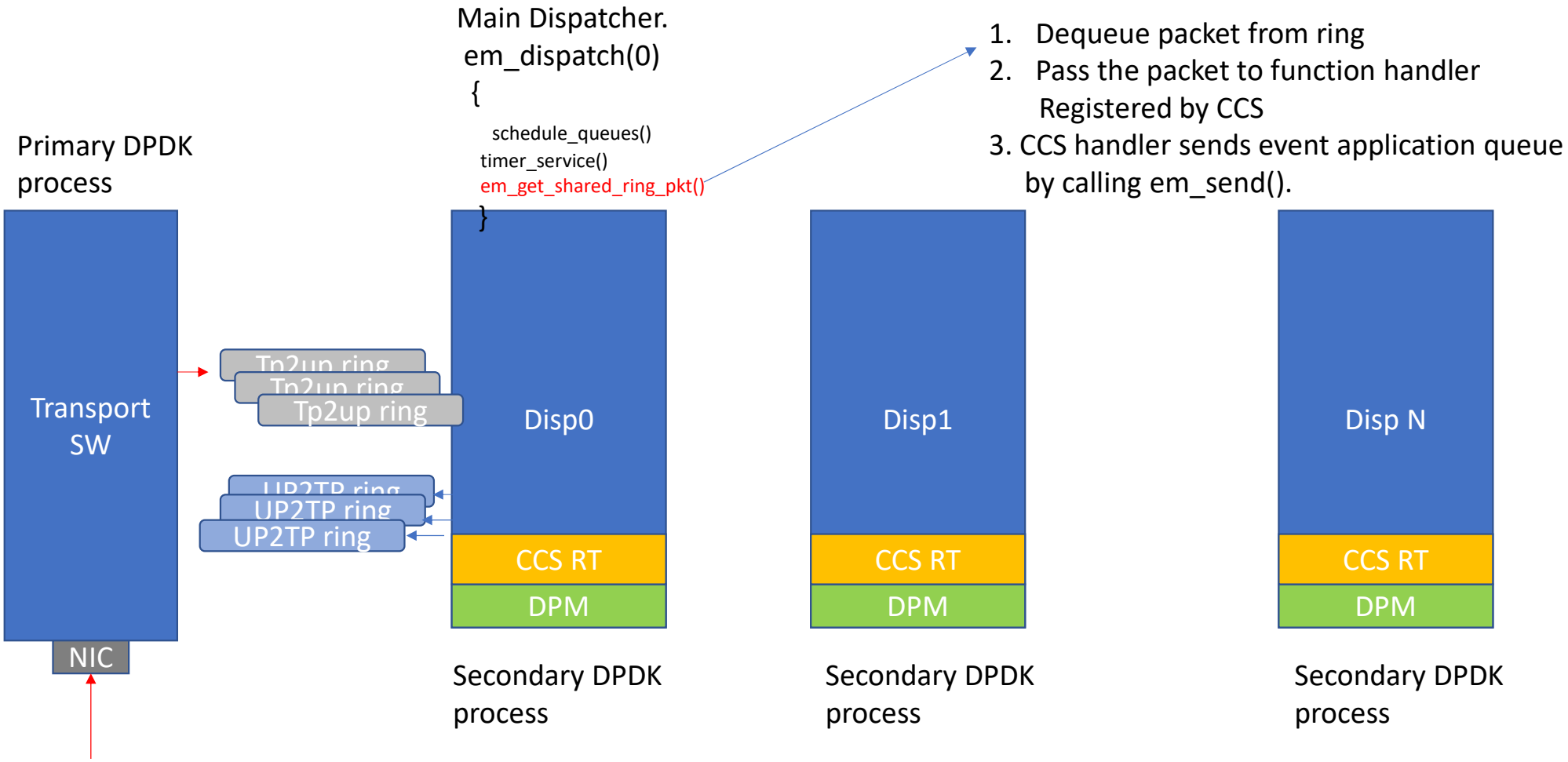
- Dequeuing single event from the queue in one schedule, is not cost effective.
- In one schedule, DPM dispatcher dequeues events in burst.
- Maximum events dequeuing is dependent upon queue priority.
- Highest priority can process maximum of 32 events in one schedule.

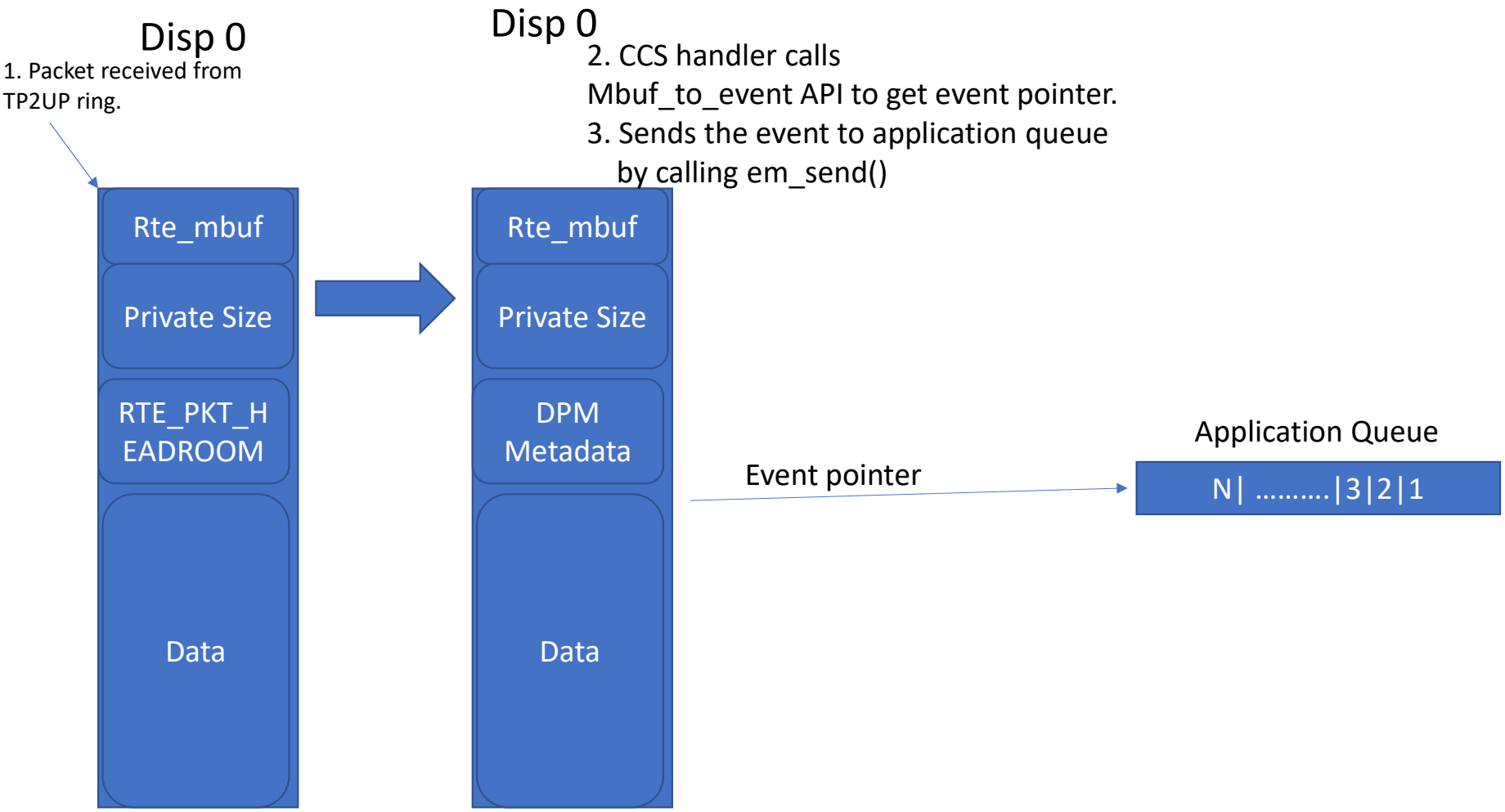
DPM event machine in Use by ASBTS

Dispatchers functionality in ASBTS



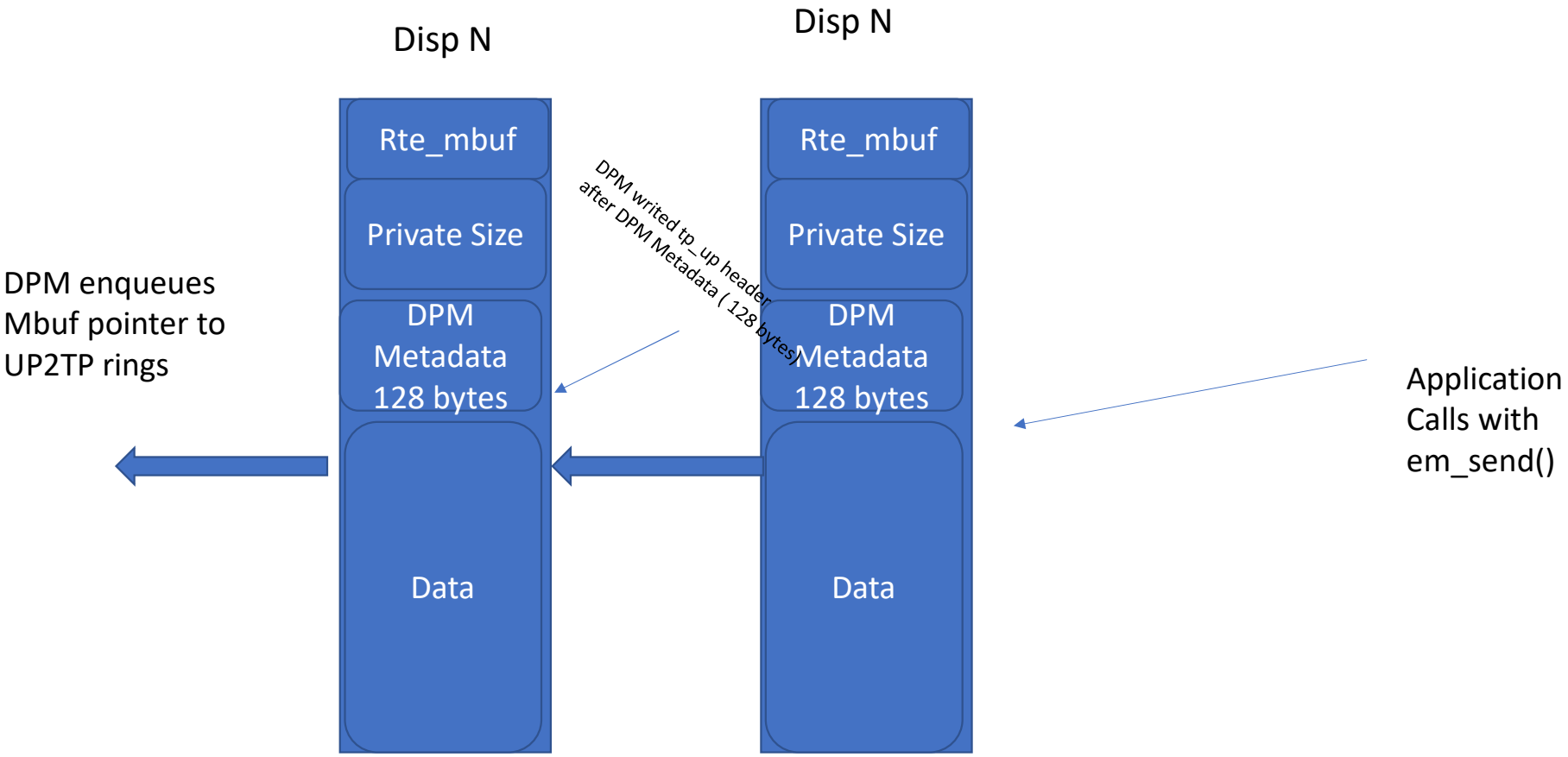
Ingress Packet flow w.r.t function


Main Dispatcher.
em_dispatch(0)
{
schedule_queues()
timer_service()
em_get_shared_ring_pkt()
}
1. Dequeue packet from ring
2. Pass the packet to function handler
Registered by CCS
3. CCS handler sends event application queue
by calling em_send().
Primary DPDK
process
Transport
SW
NIC
Tp2up ring
Tp2up ring
Tp2up ring
UP2TP ring
UP2TP ring
UP2TP ring
Disp0
CCS RT
DPM
Disp1
CCS RT
DPM
Disp N
CCS RT
DPM
Secondary DPDK
process
Secondary DPDK
process
Secondary DPDK
process
Ingress/egress Packet


Disp 0
1. Packet received from TP2UP ring.
Rte_mbuf
Private Size
RTE_PKT_HEADROOM
Data
Disp 0
2. CCS handler calls Mbuf_to_event API to get event pointer.
3. Sends the event to application queue by calling em_send()
Rte_mbuf
Private Size
DPM Metadata
Data
Event pointer
Application Queue
N| ……|3|2|1

Egress packet flow

- Application calls em_send()
- DPM checks for specific event type/queue Id
- Depending on the check, it enqueues packet to application queue or UP2TP rings


Disp N
Disp N
Rte_mbuf
Private Size
DPM Metadata 128 bytes
Data
Rte_mbuf
Private Size
DPM Metadata 128 bytes
Data
DPM writed tp_up header after DPM Metadata (128 bytes)
DPM enqueues Mbuf pointer to UP2TP rings
Application Calls with em_send()

DPM test application

- SS_TestRCPDPM/DPM_FT/testapp/em_test_proc_mode
  is test application written, which resembles DPM event machine usage by CCS RT in ASBTS.
- Main function forks and primary DPDK process is created, which acts like TRS
- Main function forks 2 more processes , which are secondary DPDK processes , act like Event Machine Dispatchers
- SS_TestRCPDPM/DPM_FT/testcli is used to send test events to Dispatchers

Performance Results

Following APIs are measured

- Em_alloc
- Em_free
- Em_send
- Number of events per core

DPM API Perf Results

Setup and Build details

HW: OR18 setup
CPU Frequency: 2095.074 MHz
RCP Build Used: RCP2.0_18.41.0

Kernel boot options used:
Isolcpus, rcu_nocbs

Em_alloc

Buffer size	no of pkts		min time	max time	avg time
1024	1000 times allocated and time calculated per allocation		156	3206	400
			162	1668	402
			164	1760	392
			152	1786	394
			154	17196	408
			172	1690	406
			172	2152	400
			150	16276	417
			148	2024	396
			150	1886	388

```
case MEM_ALLOC_PERF:
        for(int i=0;i<1000;i++)
        {
                start_time= rte_rdtsc();
                em_event_t *test_ptr=em_alloc(size,EM_PERFORMANCE_TEST_REQ,APP_DATA_POOL);
                end_time= rte_rdtsc();

                if( test_ptr == NULL )
                {
                        syslog(LOG_ERR,"memory allocation for sending em parallel events failed\n");
                        for( ;i > 0; --i)
                        {
                                em_free(parr[i]);
                        }
                        em_free((void *)event_data);
                        return;
                }
                parr[i] = test_ptr;
                diff_val = end_time - start_time;
                if( alloc_min > diff_val )
                {
                        alloc_min =diff_val;
                }
                if( alloc_max < diff_val )
                {
                        alloc_max = diff_val;
                }
                alloc_total+=diff_val;

        }
        alloc_avg = alloc_total/1000;
        syslog(LOG_ERR," in function %s, line %d, alloc min =%lu, max =%lu, avg =%lu\n",__func__,__LINE__,alloc_min,alloc_max,alloc_avg);
        free_total =0;
        for(int i=0;i<1000;i++)
        {
                start_time= rte_rdtsc();
                em_free(parr[i]);
                end_time = rte_rdtsc();
                diff_val = end_time - start_time;
                if( free_min > diff_val )
                {
                        free_min =diff_val;
                }
                if(free_max < diff_val )
                {
                        free_max = diff_val;
```

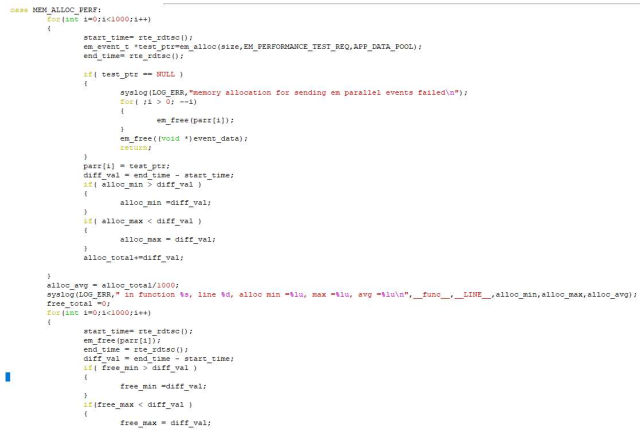
Measurement Unit in Cycle.

em_alloc cost in ns: 156/2100000000=74.28ns

Em_free

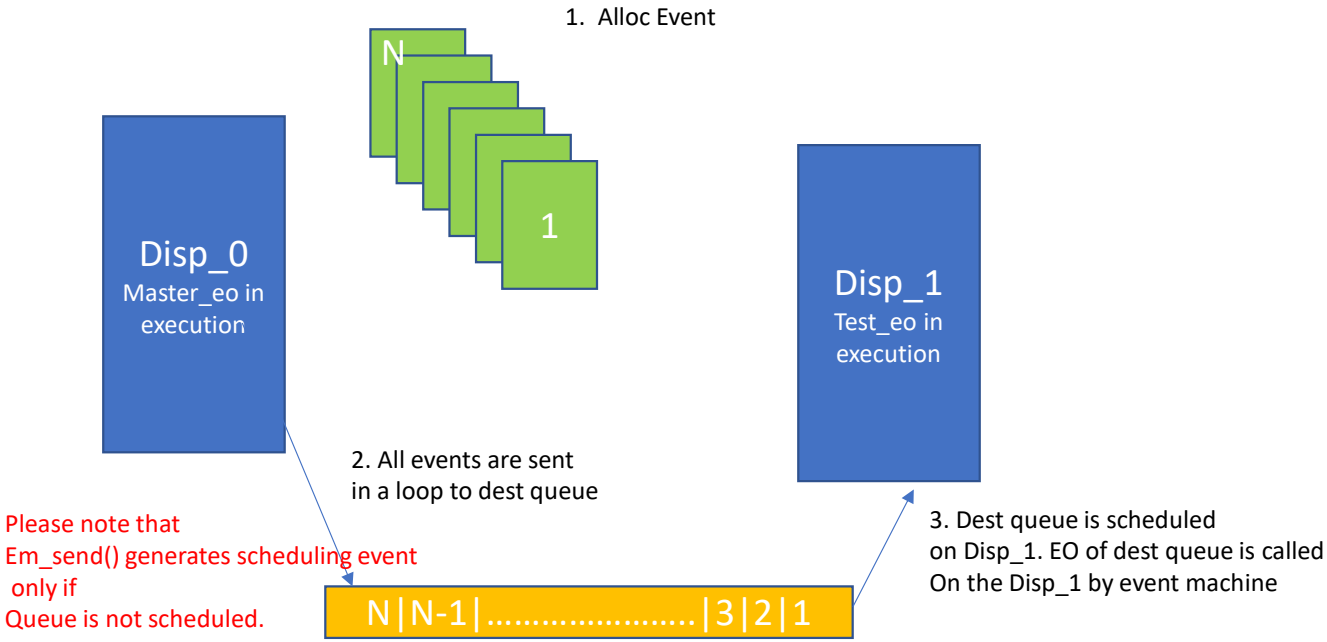
1024	1000 times allocated and time calculated per allocation		72	668	98
			70	994	100
			68	968	97
			70	928	97
			70	1080	99
			70	1484	99
			62	1008	98
			68	998	99
			68	984	100
			68	1358	99

Code snippet



```
case MEM_ALLOC_PERF:
        for(int i=0;i<1000;i++)
        {
                start_time= rte_rdtsc();
                em_event_t *test_ptr=em_alloc(size,EM_PERFORMANCE_TEST_REQ,APP_DATA_POOL);
                end_time= rte_rdtsc();

                if( test_ptr == NULL )
                {
                        syslog(LOG_ERR,"memory allocation for sending em parallel events failed\n");
                        for( ;i > 0; --i)
                        {
                                em_free(parr[i]);
                        }
                        em_free((void *)event_data);
                        return;
                }
                parr[i] = test_ptr;
                diff_val = end_time - start_time;
                if( alloc_min > diff_val )
                {
                        alloc_min =diff_val;
                }
                if( alloc_max < diff_val )
                {
                        alloc_max = diff_val;
                }
                alloc_total+=diff_val;

        }
        alloc_avg = alloc_total/1000;
        syslog(LOG_ERR," in function %s, line %d, alloc min =%lu, max =%lu, avg =%lu\n",__func__,__LINE__,alloc_min,alloc_max,alloc_avg);
        free_total =0;
        for(int i=0;i<1000;i++)
        {
                start_time= rte_rdtsc();
                em_free(parr[i]);
                end_time = rte_rdtsc();
                diff_val = end_time - start_time;
                if( free_min > diff_val )
                {
                        free_min =diff_val;
                }
                if(free_max < diff_val )
                {
                        free_max = diff_val;
```

Em_send


1. Alloc Event
N
1
Disp_0
Master_eo in execution
Disp_1
Test_eo in execution
2. All events are sent in a loop to dest queue
3. Dest queue is scheduled on Disp_1. EO of dest queue is called On the Disp_1 by event machine
Please note that Em_send() generates scheduling event only if Queue is not scheduled.
N|N-1|.....|3|2|1

Measurement

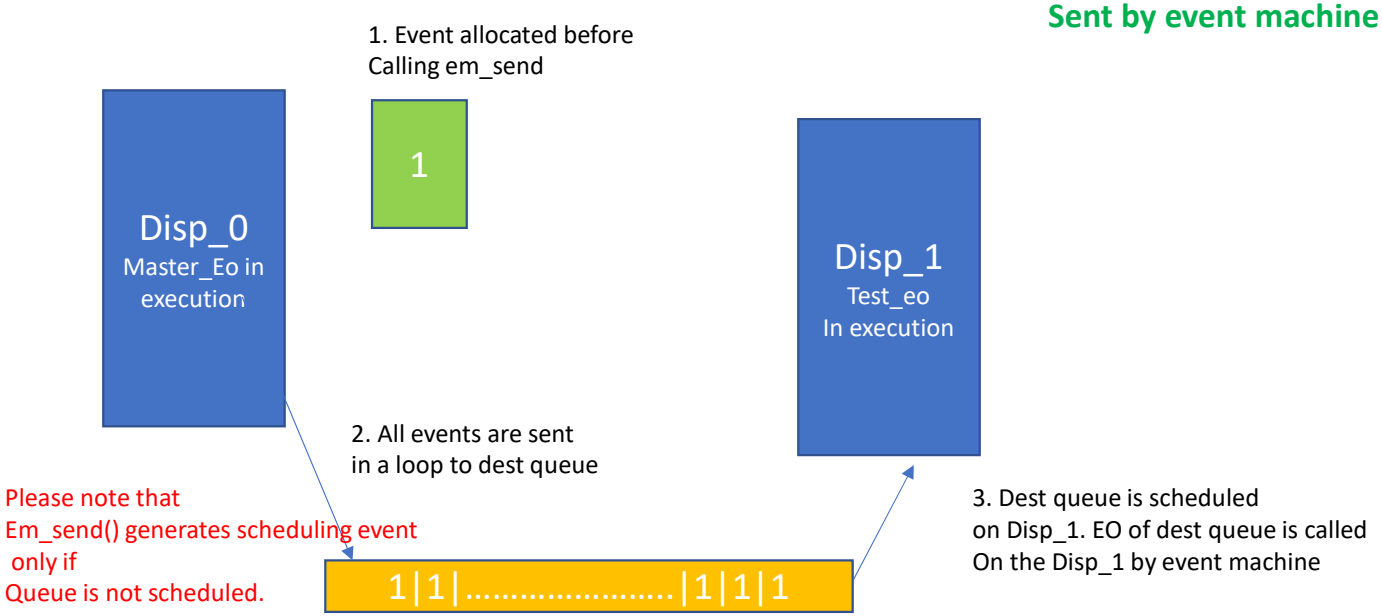
Event Size	Number of events sent		Min	Max	Avg
1024	1000		134	1266	316
	1000		142	1310	364
	1000		140	1200	358
	1000		144	1554	367
	1000		142	1264	359
	1000		138	1638	363
	1000		134	1454	367
	1000		138	1276	365
	1000		142	1320	367
	1000		136	1236	359
	1000		152	1152	357

Code Snippet

Em_send

Same event is sent to destination queue
Destination EO does not free the event unless
Free flag is set

This is to measure, rate of events that can be
Sent by event machine if dest EO can process



Measurement

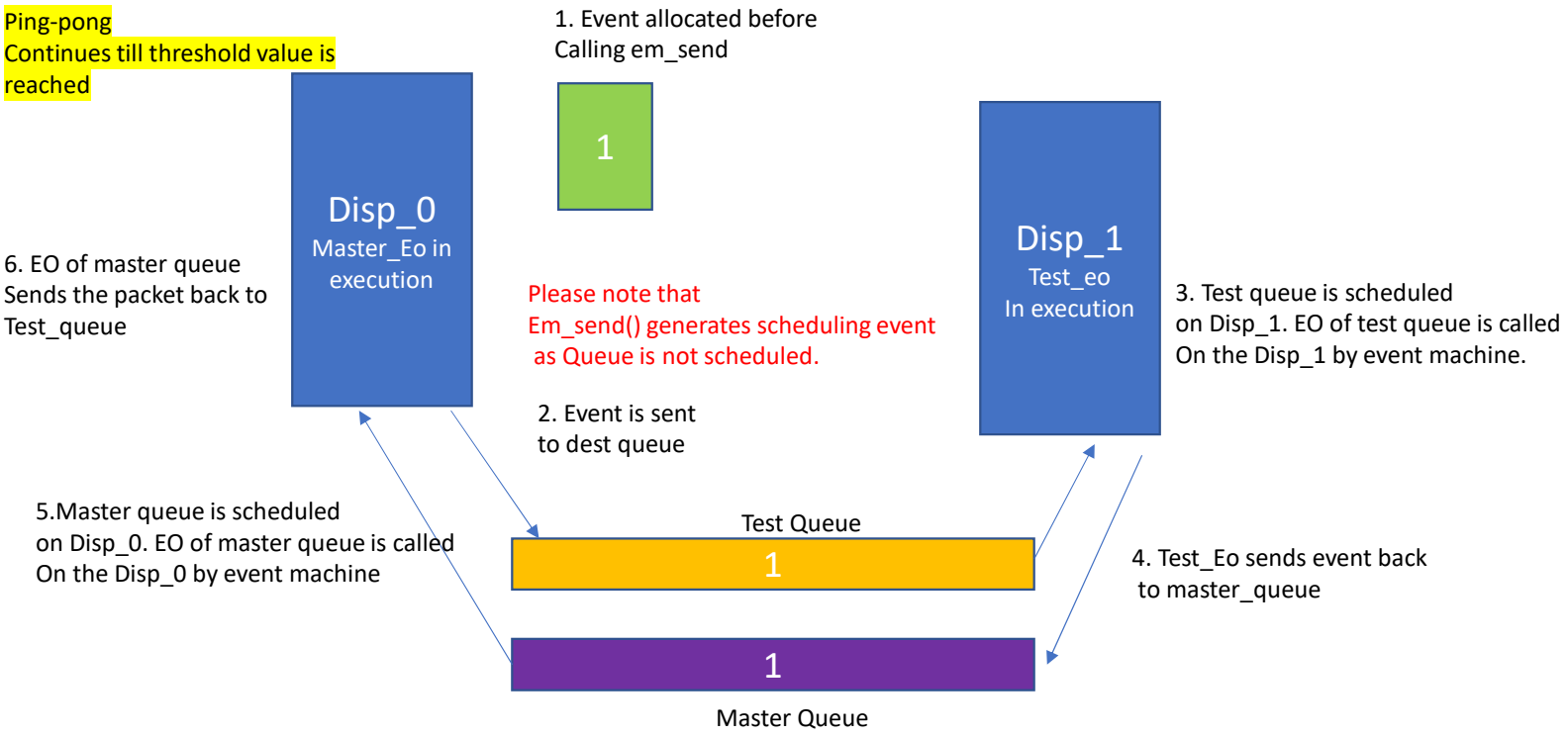
					Avg
1024	for 1000 pkts with delay 10us for every 10 pkts				157
					157
					157
					156
					157
					157
					158
					156
					157
					158
					156
					Avg
	for 1000 pkts with delay 10us for every 100 pkts				
					123
					123
					123
					123
					123
					122
					123
					123
					123
					123

Code snippet

```
                ------
case SEND_PKTS_IN_ONE_GO:
        {
        uint64_t outer_forloop_start_time =0;
        uint64_t outer_forloop_end_time=0;
        size= sizeof(event_data_t)+sizeof(em_performance_test_ack_t);
        void *ptr=em_alloc(size,TRANSFORM_MODE|EM_PERFORMANCE_TEST_REQ,1);
        if( NULL == ptr)
        {
                em_free((void *)event_data);
                syslog(LOG_ERR,"Memory allocation for Ack Msg in DPM_MEM_TEST failed");
                return;
        }
        send_min = 0xffff,send_max =0, send_avg=0, alloc_total=0 ;
        event_data_t *event=(event_data_t*)em_event_pointer(ptr);
        event->event_type=EM_PERFORMANCE_TEST_REQ;
        em_performance_test_ack_t *reply=(em_performance_test_ack_t *)(event->data);
        reply->status= 0;
        reply->send_pkt_results = 0;
        reply->mode= SEND_PKTS_IN_ONE_GO;

                outer_forloop_start_time= rte_rdtsc();
        for(int i=0;i<1000;i++)
        {
        for(int j=0;j<no_of_pkts;j++)
        {

        if( em_send((void*)ptr,req->queue_id))
        {
                em_free((void *)ptr);
                syslog(LOG_ERR,"Packet send failure in %s line no %d i = %d j = %d\n",__func__,__LINE__,i,j);
        }
        }
                rte_delay_us(10);
        }
        outer_forloop_end_time= rte_rdtsc();
```

Number of events per core

This is to measure, core capacity to send Events including scheduling event generation



Measurement

no of pkts sent		start time		end time		difference		time taken/ round trip		time taken for em_send (enqueue + scheduling). This is also measure of number of events/per core in event machine
100000		3.43E+15		3.43E+15		1.94E+08		1936		968
100000		3.43E+15		3.43E+15		1.93E+08		1926		963
100000		3.43E+15		3.43E+15		1.94E+08		1940		970
100000		3.43E+15		3.43E+15		1.93E+08		1934		967
100000		3.43E+15		3.43E+15		1.93E+08		1933		967
100000		3.43E+15		3.43E+15		1.93E+08		1932		966
100000		3.43E+15		3.43E+15		1.93E+08		1932		966
100000		3.43E+15		3.43E+15		1.93E+08		1932		966
100000		3.43E+15		3.43E+15		1.93E+08		1933		967
100000		3.43E+15		3.43E+15		1.93E+08		1933		967

Code snippet

Master EO

```
void process_threshold_reply(event_data_t* event_data)
{
        event_data_t *local_ptr = em_event_pointer(event_data);
        local_ptr->event_type=EM_PERFORMANCE_TEST_REQ;
        em_performance_test_ack_t *reply=(em_performance_test_ack_t *)(local_ptr->data);
        curr_pkts_reced++;
                                reply->mode = THRESHOLD_VALUE;
        if(reply->mode == THRESHOLD_VALUE )
        {
        if(curr_pkts_reced == threshold_value)
        {
                threshold_end_time = rte_rdtsc();
                local_ptr->event_type=TRANSFORM_MODE| EM_PERFORMANCE_TEST_REPLY;
                reply->status = 1;
                reply->threshold_perf_test_ack.threshold_avg_time = (threshold_end_time - threshold_start_time)/threshold_value;
                 syslog(LOG_ERR,"test case success func is %lu:%lu:%lu:%u:%lu  \n",threshold_end_time,threshold_start_time,(threshold_end_time - threshold_start_time),thres
hold_value,reply->threshold_perf_test_ack.threshold_avg_time);
                em_send((void*)event_data,EGRESS_QID);
                curr_pkts_reced =0;
                return;
        }
        else if( em_send((void*)event_data,queue_id))
        {
                syslog(LOG_ERR,"Packet send failure in %s\n",__func__);
                return;
        }
        }
}
```

Test EO

```
case THRESHOLD_VALUE:
                local_ptr->event_type=EM_THRESHOLD_REPLY;
                if( em_send((void*)event,MASTER_QID))
                {
                        syslog(LOG_ERR,"Packet send failure in %s\n",__func__);
                        return;
                }
        break;
default:
```