Intelligent Systems

Intel® Data Plane Development Kit - Multi-process Sample Applications

User Guide

August 2012

Intel Confidential

Reference Number: 482253-1.3

Contents

Figures

Revision History

Date	Revision	Description
August 2012	1.3	Updates to Section 2.3.2 and Section 2.4.2
March 2012	1.2	Updates for software release 1.2
December 2011	1.1	Major rework and reorganization to reflect new version of sample application
September 2011	1.0	Initial version of document

1.0 Introduction

This manual describes the example applications for multi-processing that are included in the Intel® Data Plane Development Kit (Intel® DPDK).

2.0 Example Applications

2.1 Building the Sample Applications

The multi-process example applications are built in the same way as other sample applications, and as documented in the *Intel® DPDK Getting Started Guide*. To build all the example applications:

1. Set RTE_SDK and go to the example directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/multi_process
   ```

2. Set the target (a default target will be used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-default-linuxapp-gcc
   ```

   See the *Intel® DPDK Getting Started Guide* for possible `RTE_TARGET` values.

3. Build the applications:

   ```
   make
   ```

Note: If just a specific multi-process application needs to be built, the final make command can be run just in that application's directory, rather than at the top-level multi-process directory.

2.2 Basic Multi-process Example

The `examples/simple_mp` folder in the Intel® DPDK release contains a basic example application to demonstrate how two Intel® DPDK processes can work together using queues and memory pools to share information.

2.2.1 Running the Application

To run the application, start one copy of the `simple_mp` binary in one terminal, passing at least two cores in the coremask, as follows:

```
./build/simple_mp -c 3 -n 4 --proc-type=primary
```

For the first Intel® DPDK process run, the `proc-type` flag can be omitted or set to `auto`, since all Intel® DPDK processes will default to being a primary instance, meaning they have control over the hugepage shared memory regions. The process should start successfully and display a command prompt as follows:

```
$ ./build/simple_mp -c 3 -n 4 --proc-type=primary
EAL: coremask set to 3
EAL: Detected lcore 0 on socket 0
EAL: Detected lcore 1 on socket 0
EAL: Detected lcore 2 on socket 0
EAL: Detected lcore 3 on socket 0
```

```
...
EAL: Requesting 2 pages of size 1073741824
EAL: Requesting 768 pages of size 2097152
EAL: Ask a virtual area of 0x40000000 bytes
EAL: Virtual area found at 0x7ff200000000 (size = 0x40000000)
...
EAL: check igb_uio module
EAL: check module finished
EAL: Master core 0 is ready (tid=54e41820)
EAL: Core 1 is ready (tid=53b32700)
Starting core 1

simple_mp >
```

To run the secondary process to communicate with the primary process, again run the same binary setting at least two cores in the coremask.

```
    ./build/simple_mp -c C -n 4 --proc-type=secondary
```

When running a secondary process such as that shown above, the `proc-type` parameter can again be specified as `auto`. However, omitting the parameter altogether will cause the process to try and start as a primary rather than secondary process.

Once the process type is specified correctly, the process starts up, displaying largely similar status messages to the primary instance as it initializes. Once again, you will be presented with a command prompt.

Once both processes are running, messages can be sent between them using the `send` command. At any stage, either process can be terminated using the `quit` command.

```
EAL: Master core 10 is ready (tid=b5f89820)     EAL: Master core 8 is ready (tid=864a3820)
EAL: Core 11 is ready (tid=84ffe700)            EAL: Core 9 is ready (tid=85995700)
Starting core 11                                Starting core 9

simple_mp > send hello_secondary                simple_mp > core 9: Received 'hello_secondary'

simple_mp > core 11: Received 'hello_primary'
                                                simple_mp > send hello_primary

simple_mp > quit                                simple_mp > quit
```

Note: If the primary instance is terminated, the secondary instance must also be shut-down and restarted after the primary. This is necessary because the primary instance will clear and reset the shared memory regions on startup, invalidating the secondary process's pointers. The secondary process can be stopped and restarted without affecting the primary process.

2.2.2 How the Application Works

The core of this example application is based on using two queues and a single memory pool in shared memory. These three objects are created at startup by the primary process, since the secondary process cannot create objects in memory as it cannot reserve memory zones, and the secondary process then uses lookup functions to attach to these objects as it starts up.

```
if (rte_eal_process_type() == RTE_PROC_PRIMARY){
    send_ring = rte_ring_create(_PRI_2_SEC, ring_size, SOCKET0, flags);
    recv_ring = rte_ring_create(_SEC_2_PRI, ring_size, SOCKET0, flags);
    message_pool = rte_mempool_create(_MSG_POOL, pool_size,
            string_size, pool_cache, priv_data_sz,
                NULL, NULL, NULL, NULL,
                SOCKET0, flags);
} else {
    recv_ring = rte_ring_lookup(_PRI_2_SEC);
    send_ring = rte_ring_lookup(_SEC_2_PRI);
    message_pool = rte_mempool_lookup(_MSG_POOL);
}
```

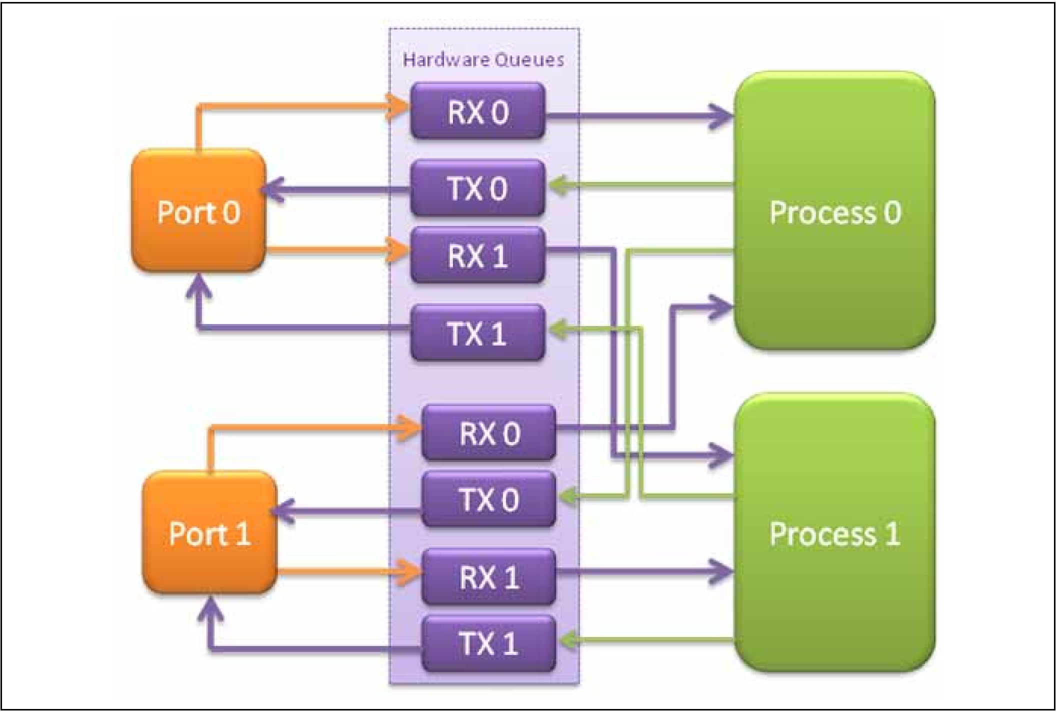
Note, however, that the named ring structure used as `send_ring` in the primary process is the `recv_ring` in the secondary process.

Once the rings and memory pools are all available in both the primary and secondary processes, the application simply dedicates two threads to sending and receiving messages respectively. The receive thread simply dequeues any messages on the receive ring, prints them, and frees the buffer space used by the messages back to the memory pool. The send thread makes use of the command-prompt library to interactively request user input for messages to send. Once a `send` command is issued by the user, a buffer is allocated from the memory pool, filled in with the message contents, then enqueued on the appropriate rte_ring.

2.3 Symmetric Multi-process Example

The second example of Intel® DPDK multi-process support demonstrates how a set of processes can run in parallel, with each process performing the same set of packet-processing operations. (Since each process is identical in functionality to the others, we refer to this as symmetric multi-processing, to differentiate it from asymmetric multi-processing - such as a client-server mode of operation seen in the next example, where different processes perform different tasks, yet co-operate to form a packet-processing system.) The following diagram shows the data-flow through the application, using two processes.

Figure 1. Example Data Flow in a Symmetric Multi-process Application



As the diagram shows, each process reads packets from each of the network ports in use. RSS is used to distribute incoming packets on each port to different hardware RX queues. Each process reads a different RX queue on each port and so does not contend with any other process for that queue access. Similarly, each process writes outgoing packets to a different TX queue on each port.

2.3.1 Running the Application

As with the `simple_mp` example, the first instance of the `symmetric_mp` process must be run as the primary instance, though with a number of other application-specific parameters also provided after the EAL arguments. These additional parameters are:

- `-p <portmask>`, where `portmask` is a hexadecimal bitmask of what ports on the system are to be used. For example: `-p 3` to use ports 0 and 1 only.
- `--num-procs <N>`, where `N` is the total number of `symmetric_mp` instances that will be run side-by-side to perform packet processing. This parameter is used to configure the appropriate number of receive queues on each network port.
- `--proc-id <n>`, where `n` is a numeric value in the range 0 <= n < N (number of processes, specified above). This identifies which `symmetric_mp` instance is being run, so that each process can read a unique receive queue on each network port.

The secondary `symmetric_mp` instances must also have these parameters specified, and the first two must be the same as those passed to the primary instance, or errors result.

For example, to run a set of four `symmetric_mp` instances, running on lcores 1-4, all performing level-2 forwarding of packets between ports 0 and 1, the following commands can be used (assuming run as root):

```
# ./build/symmetric_mp -c 2 -n 4 --proc-type=auto -- -p 3 --num-procs=4 --proc-id=0
# ./build/symmetric_mp -c 4 -n 4 --proc-type=auto -- -p 3 --num-procs=4 --proc-id=1
# ./build/symmetric_mp -c 8 -n 4 --proc-type=auto -- -p 3 --num-procs=4 --proc-id=2
# ./build/symmetric_mp -c 10 -n 4 --proc-type=auto -- -p 3 --num-procs=4 --proc-id=3
```

Note: In the above example, the process type can be explicitly specified as `primary` or `secondary`, rather than `auto`. When using `auto`, the first process run creates all the memory structures needed for all processes - irrespective of whether it has a `proc-id` of 0, 1, 2 or 3.

Note: For the symmetric multi-process example, since all processes work in the same manner, once the hugepage shared memory and the network ports are initialized, it is not necessary to restart all processes if the primary instance dies. Instead, that process can be restarted as a secondary, by explicitly setting the `proc-type` to `secondary` on the command line. (All subsequent instances launched will also need this explicitly specified, as auto-detection will detect no primary processes running and therefore attempt to re-initialize shared memory.)

2.3.2 How the Application Works

The initialization calls in both the primary and secondary instances are the same for the most part, calling the `rte_eal_init()`, 1G and 10 G driver initialization and then `rte_eal_pci_probe()` functions. Thereafter, the initialization done depends on whether the process is configured as a primary or secondary instance.

In the primary instance, a memory pool is created for the packet `mbufs` and the network ports to be used are initialized - the number of RX and TX queues per port being determined by the `num-procs` parameter passed on the command-line. The structures for the initialized network ports are stored in shared memory and therefore will be accessible by the secondary process as it initializes.

```
if (num_ports & 1)
    rte_exit(EXIT_FAILURE, "Application must use an even number of ports\n");
for(i = 0; i < num_ports; i++){
    if(proc_type == RTE_PROC_PRIMARY)
        if (smp_port_init(ports[i], mp, (uint16_t)num_procs) < 0)
            rte_exit(EXIT_FAILURE, "Error initialising ports\n");
}
```

In the secondary instance, rather than initializing the network ports, the port information exported by the primary process is used, giving the secondary process access to the hardware and software rings for each network port. Similarly, the memory pool of `mbufs` is accessed by doing a lookup for it by name:

```
mp = (proc_type == RTE_PROC_SECONDARY) ?
        rte_mempool_lookup(_SMP_MBUF_POOL) :
        rte_mempool_create(_SMP_MBUF_POOL, NB_MBUFS, MBUF_SIZE, ... )
```

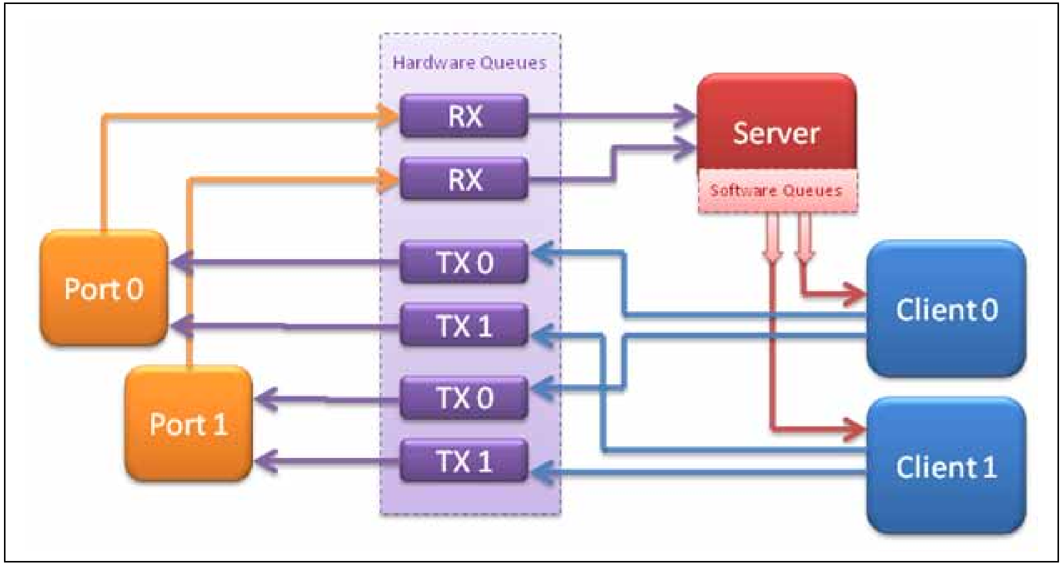
Once this initialization is complete, the main loop of each process, both primary and secondary, is exactly the same - each process reads from each port using the queue corresponding to its `proc-id` parameter, and writes to the corresponding transmit queue on the output port.

2.4 Client-Server Multi-process Example

The third example multi-process application included with the Intel® DPDK shows how one can use a client-server type multi-process design to do packet processing. In this example, a single server process performs the packet reception from the ports being used and distributes these packets using round-robin ordering among a set of client processes, which perform the actual packet processing. In this case, the client applications just perform level-2 forwarding of packets by sending each packet out on a different network port.

The following diagram shows the data-flow through the application, using two client processes.

Figure 2. Example Data Flow in a Client-Server Symmetric Multi-process Application



2.4.1 Running the Application

The server process must be run initially as the primary process to set up all memory structures for use by the clients. In addition to the EAL parameters, the application-specific parameters are:

- `-p <portmask>`, where `portmask` is a hexadecimal bitmask of what ports on the system are to be used. For example: `-p 3` to use ports 0 and 1 only.
- `-n <num-clients>`, where the `num-clients` parameter is the number of client processes that will process the packets received by the server application.

Note: In the server process, a single thread, the master thread, that is, the lowest numbered lcore in the coremask, performs all packet I/O. If a coremask is specified with more than a single lcore bit set in it, an additional lcore will be used for a thread to periodically print packet count statistics.

Since the server application stores configuration data in shared memory, including the network ports to be used, the only application parameter needed by a client process is its client instance ID. Therefore, to run a server application on lcore 1 (with lcore 2 printing statistics) along with two client processes running on lcores 3 and 4, the following commands could be used:

```
# ./mp_server/build/mp_server -c 6 -n 4 -- -p 3 -n 2
# ./mp_client/build/mp_client -c 8 -n 4 --proc-type=auto -- -n 0
# ./mp_client/build/mp_client -c 10 -n 4 --proc-type=auto -- -n 1
```

Note: If the server application dies and needs to be restarted, all client applications also need to be restarted, as there is no support in the server application for it to run as a secondary process. Any client processes that need restarting can be restarted without affecting the server process.

2.4.2 How the Application Works

The server process performs the network port and data structure initialization much as the symmetric multi-process application does when run as primary. One additional enhancement in this sample application is that the server process stores its port configuration data in a memory zone in hugepage shared memory. This eliminates the need for the client processes to have the `portmask` parameter passed into them on the command line, as is done for the symmetric multi-process application, and therefore eliminates mismatched parameters as a potential source of errors.

In the same way that the server process is designed to be run as a primary process instance only, the client processes are designed to be run as secondary instances only. They have no code to attempt to create shared memory objects. Instead, handles to all needed rings and memory pools are obtained via calls to `rte_ring_lookup()` and `rte_mempool_lookup()`. The network ports for use by the processes are obtained by loading the network port drivers and probing the PCI bus, which will, as in the symmetric multi-process example, automatically get access to the network ports using the settings already configured by the primary/server process.

Once all applications are initialized, the server operates by reading packets from each network port in turn and distributing those packets to the client queues (software rings, one for each client process) in round-robin order. On the client side, the packets are read from the rings in as big of bursts as possible, then routed out to a different network port. The routing used is very simple. All packets received on the first NIC port are transmitted back out on the second port and vice versa. Similarly, packets are routed between the 3rd and 4th network ports and so on. The sending of packets is done by writing the packets directly to the network ports; they are not transferred back via the server process.

In both the server and the client processes, outgoing packets are buffered before being sent, so as to allow the sending of multiple packets in a single burst to improve efficiency. For example, the client process will buffer packets to send, until either the buffer is full or until we receive no further packets from the server.

§ §