F-Stack 研究文档

目录

1 F-Stack 介绍

F-Stack 是一个全用户态的高性能的网络接入开发包，基于 DPDK、FreeBSD 协议栈、微线程接口等，适用于各种需要网络接入的业务，用户只需要关注业务逻辑，简单的接入 F-Stack 即可实现高性能的网络服务器。

1.1 内核协议栈面临的瓶颈

随着网卡性能的飞速发展，10GE 网卡已经大规模普及，25GE/40GE/100GE 网卡也在逐步推广，linux 内核在网络数据包处理上的瓶颈也越发明显，在传统的内核协议栈中，网卡通过硬件中断通知协议栈有新的数据包到达，内核的网卡驱动程序负责处理这个硬件中断，将数据包从网卡队列拷贝到内核开辟的缓冲区中（DMA），然后数据包经过一系列的协议处理流程，最后送到用户程序指定的缓冲区中。在这个过程中中断处理、内存拷贝、系统调用（锁、软中断、上下文切换）等严重影响了网络数据包的处理能力。操作系统的对应用程序和数据包处理的调度可能跨 CPU 调度，局部性失效进一步影响网络性能。

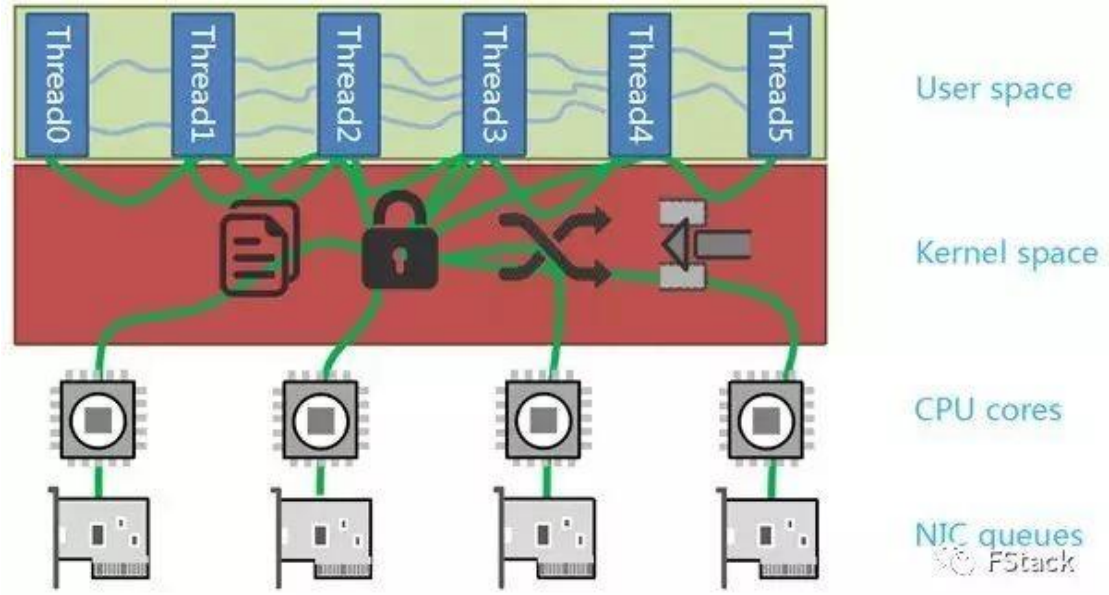

图 1-1 传统内核协议栈的现状

1.2 解决方案

而互联网的快速发展亟需高性能的网络处理能力，kernel bypass 方案也越来被人所接受，市场上也出现了多种类似技术，如 DPDK、NETMAP、PF_RING 等，其核心思想就是内核只用来处理控制流，所有数据流相关操作都在用户态进行处理，从而规避内核的包拷贝、线程调度、系统调用、中断等性能瓶颈，并辅以各种性能调优手段，从而达到更高的性能。其中 DPDK 因为更彻底的脱离内核调度以及活跃的社区支持从而得到了更广泛的使用。

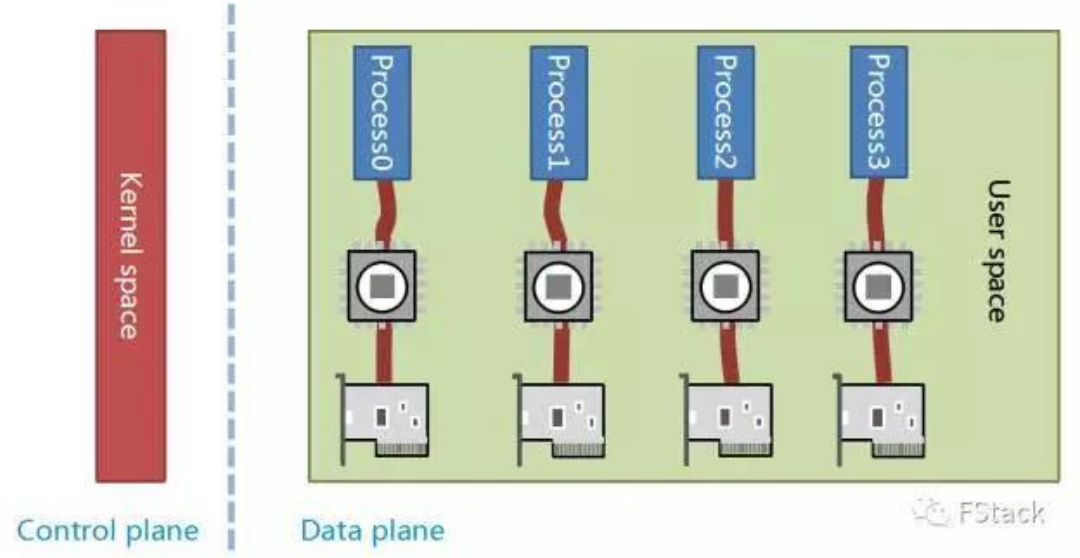

图 1-2 DPDK kernel bypass 调度

1.3 F-Stack 开发框架

F-Stack 是一款兼顾高性能、易用性和通用性的网络开发框架，传统上 DPDK 大多用于 SDN、NFV、DNS 等简单的应用场景下，对于复杂的 TCP 协议栈上的七层应用很少，市面上已出现了部分用户态协议栈，如 mTCP、Mirage、lwIP、NUSE 等，也有用户态的编程框架，如 SeaStar[注 1]等，但统一的特点是应用程序接入门槛较高，不易于使用。

F-Stack 使用纯 C 实现，充当胶水粘合了 DPDK、FreeBSD 用户态协议栈、Posix API、微线程框架和上层应用（Nginx、Redis），使绝大部分的网络应用可以通过直接修改配置或替换系统的网络接口即可接入 F-Stack，从而获得更高的网络性能。

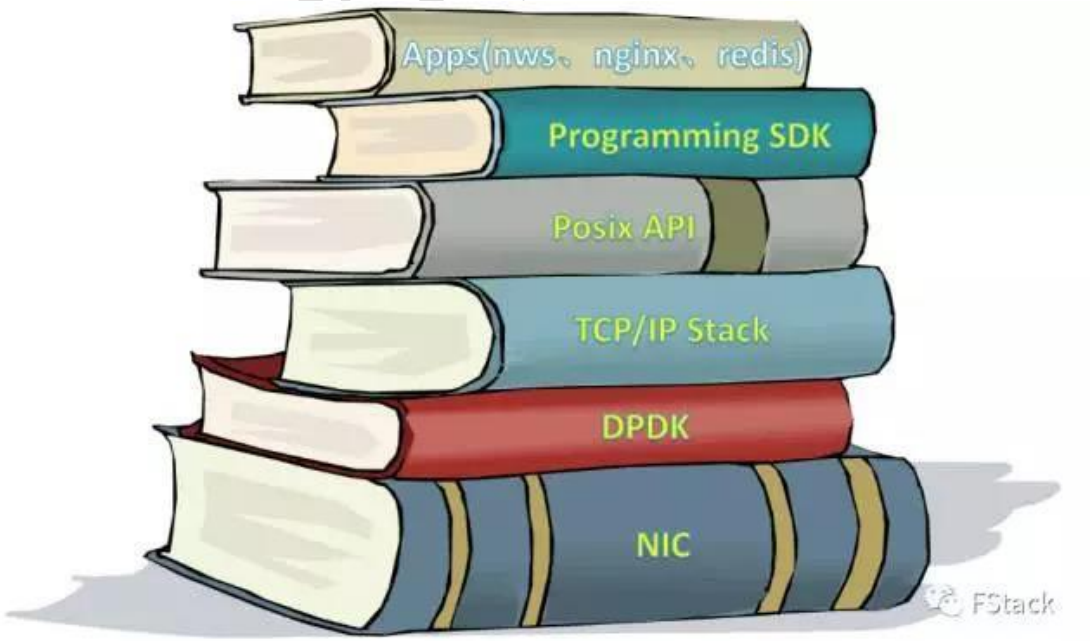

图 1-3 F-Stack 架构

F-Stack 总体架构如图 1-3 所示，并使用多进程架构，如以下特点。

- 使用多进程无共享架构。

  各进程绑定独立的网卡队列和 CPU，请求通过设置网卡 RSS 散落到各进程进行处理。

  各进程拥有独立的协议栈、PCB 表等资源。

  每个 NUMA 节点使用独立的内存池。

  进程间通信通过无锁环形队列（rte_ring）进行。
- 使用 DPDK 作为网络 I/O 模块，将数据包从网卡直接接收到用户态。
- 移植 FreeBSD 11.0.1 协议栈到用户态并与 DPDK 对接。

2 运行环境

目前 F-Stack 的运行环境标准如下表所示：

操作系统	Linux
发行版本	Redhat/CentOS 7.0+
内核版本	3.10.0+
GCC 版本	4.8.0+
网卡要求	参考 DPDK 的[list of supported NICs] (http://dpdk.org/doc/nics)

3 编译 F-Stack

3.1 准备工作

1. 修改/usr/src/kernels/3.10.0-862.el7.x86_64/include/linux/netdevice.h 文件内容如下所示：

```
1295        int                         (*ndo_set_config)(struct net_device *dev,
1296                                                      struct ifmap *map);
1297        /* BEGIN: Modified by zhangcan, 2018/6/12 */
1298        //RH_KABI_RENAME(int        (*ndo_change_mtu),
1299        //                  int       (*ndo_change_mtu_rh74))(struct net_device *dev,
1300        //                                                    int new_mtu);
1301        int                         (*ndo_change_mtu)(struct net_device *dev,
1300                                                      int new_mtu);
1302        /* END:    Modified by zhangcan, 2018/6/12 */
1303        int                         (*ndo_neigh_setup)(struct net_device *dev,
1304                                                      struct neigh_parms *);
```

要将 RH_KABI_RENAME 宏定义的 ndo_change_mtu 函数替换掉。

2. 修改/dpdk/lib/librte_eal/linuxapp/igb_uio/igb_uio.c 文件内容如下所示：

```
277             /* fall back to INTX */
278             case RTE_INTR_MODE_LEGACY:
279                     /* BEGIN: Modified by zhangcan, 2018/6/13 */
280 #if 0
281                     if (pci_intx_mask_supported(udev->pdev)) {
282 #else
283                     if (pci_intx_mask_supported(udev->pdev) || true) {
284 #endif
285                     /* END:     Modified by zhangcan, 2018/6/13 */
286                             dev_dbg(&udev->pdev->dev, "using INTX");
287                                     udev->info.irq_flags = IRQF_SHARED |
IRQF_NO_THREAD;
288                             udev->info.irq = udev->pdev->irq;
289                             udev->mode = RTE_INTR_MODE_LEGACY;
290                             break;
291                     }
292                     dev_notice(&udev->pdev->dev, "PCI INTX mask not supported\n");
```

在虚拟机运行程序时接口 pci_intx_mask_supported 获取会失败，所以要将 if 判断改为 true。

3.2 编译源码

◆ 源码下载

```
https://github.com/F-Stack/f-stack/releases/tag/v1.12
```

◆ 编译 DPDK

```
cd dpdk/usertools
./dpdk-setup.sh # compile with x86_64-native-linuxapp-gcc
```

◆ 设置大叶内存

```
# single-node system
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages

# or NUMA
echo 1024 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
echo 1024 > /sys/devices/system/node/node1/hugepages/hugepages-2048kB/nr_hugepages
```

◆ 使用大叶内存

```
mkdir /mnt/huge
mount -t hugetlbfs nodev /mnt/huge
```

◆ 加载网卡驱动

```
modprobe uio
insmod /dpdk/x86_64-native-linuxapp-gcc/kmod/igb_uio.ko
insmod /dpdk/x86_64-native-linuxapp-gcc/kmod/rte_kni.ko
```

```
ifconfig eth0 down
python dpdk-devbind.py --bind=igb_uio eth0
```

◆ 编译 F-Stack

```
export FF_PATH=/data/f-stack
export FF_DPDK=/data/f-stack/dpdk/x86_64-native-linuxapp-gcc
cd ../../lib/
make
```

◆ 编译 nginx

```
cd app/nginx-1.11.10
./configure --prefix=/usr/local/nginx_fstack --with-ff_module
make
make install
[注]：如果 nginx 不适用 F-Stack，编译时可不使能--with-ff_module
```

4 配置文件

nginx 安装路径的 conf 目录 f-stack.conf 配置文件。

```
[dpdk]
## Hexadecimal bitmask of cores to run on.
#lcore_mask=3
# /* BEGIN: Added by zhangcan, 2018/8/7 */
llcore_list=0,1
# /* END:     Added by zhangcan, 2018/8/7 */
channel=4
promiscuous=1
numa_on=1
## TCP segment offload, default: disabled.
tso=0
## HW vlan strip, default: enabled.
vlan_strip=1

# /* BEGIN: Added by zhangcan, 2018/7/26 */
control_core=2
# /* END:     Added by zhangcan, 2018/7/26 */

# enabled port list
#
# EBNF grammar:
#
#       exp          ::= num_list {"," num_list}
#       num_list ::= <num> | <range>
```

```
#       range       ::= <num>"-"<num>
#       num         ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
#
# examples
#       0-3           ports 0, 1,2,3 are enabled
#       1-3,4,7     ports 1,2,3,4,7 are enabled
port_list=0

## Port config section
## Correspond to dpdk.port_list's index: port0, port1...
[port0]
addr=192.190.20.163
netmask=255.255.255.0
broadcast=192.190.20.255
gateway=192.190.20.1
lcore_list=0

## Packet capture path, this will hurt performance
#pcap=./a.pcap

## Kni config: if enabled and method=reject,
## all packets that do not belong to the following tcp_port and udp_port
## will transmit to kernel; if method=accept, all packets that belong to
## the following tcp_port and udp_port will transmit to kernel.
[kni]
#enable=1
#method=reject
## The format is same as port_list
#tcp_port=80

## FreeBSD network performance tuning configurations.
## Most native FreeBSD configurations are supported.
[freebsd.boot]
hz=100

## Block out a range of descriptors to avoid overlap
## with the kernel's descriptor space.
## You can increase this value according to your app.
fd_reserve=1024

kern.ipc.maxsockets=262144

net.inet.tcp.syncache.hashsize=4096
net.inet.tcp.syncache.bucketlimit=100
```

```
net.inet.tcp.tcbhashsize=65536

kern.ncallout=262144

[freebsd.sysctl]
kern.ipc.somaxconn=32768
kern.ipc.maxsockbuf=16777216

net.link.ether.inet.maxhold=5

net.inet.tcp.fast_finwait2_recycle=1
net.inet.tcp.sendspace=16384
net.inet.tcp.recvspace=8192
net.inet.tcp.nolocaltimewait=1
net.inet.tcp.cc.algorithm=cubic
net.inet.tcp.sendbuf_max=16777216
net.inet.tcp.recvbuf_max=16777216
net.inet.tcp.sendbuf_auto=1
net.inet.tcp.recvbuf_auto=1
net.inet.tcp.sendbuf_inc=16384
net.inet.tcp.recvbuf_inc=524288
net.inet.tcp.sack.enable=1
net.inet.tcp.blackhole=1
net.inet.tcp.msl=2000
#net.inet.tcp.delayed_ack=0

net.inet.udp.blackhole=0
net.inet.ip.redirect=0

# /* BEGIN: Added by zhangcan, 2018/7/26 */
net.inet.ip.forwarding=1
# /* END:     Added by zhangcan, 2018/7/26 */
```

具体字段描述如下所示：

[dpdk]	**dpdk 相关配置选项**
lcore_mask	逻辑核 16 进制掩码，要和 nginx 配置文件进程数相同
llcore_list	[新增] 逻辑核 10 进制列表，与 lcore_mask 相同，两者选其一
channel	每个处理器 socket 的内存通道数量
promiscuous	设置网卡混杂模式开关
numa_on	设置 numa 开关
control_core	[新增] 配置控制核，用于配置更新或其它作用而设计
port_list	端口列表，可以配置多个
[port0]	对应端口列表配置的端口 port0
addr	IP 地址

netmask	掩码
broadcast	广播地址
gateway	网关
lcore_list	该端口使能的逻辑核
pcap	设置抓包路径
[kni]	**kni 相关配置选项**
enable	参数配置是否开启 KNI，设置为 1 表示开启，设置为 0 表示关闭，默认值为 0
method	本参数和 tcp_port/udp_port 参数配合使用，method 参数可选 accept 或 reject 当 method 参数设置为 accept 时，默认所有数据包都交由 F-Stack 处理，只将 tcp_port/udp_port 参数指定的端口的数据包通过 KNI 转发至系统内核。 当 method 参数设置为 reject 时，默认素有的数据包都通过 KNI 转发至系统内核，只将 tcp_port/udp_port 参数指定的端口号的数据包交由 F-Stack 处理。
tcp_port	指定 KNI 处理的端口号，如有多个端口号用逗号","分隔，多个相连端口号也可以同时设置，如 80-90
[freebsd.boot] **[freebsd.sysctl]**	**FreeBSD 网络性能调优配置选项**
net.inet.ip.forwarding	[新增] 是否开启转发功能

5 处理流程

5.1 多进程模式

F-Stack 上层应用采用多进程模式，第一个 worker 进程 prco_id 为 0，且为 primary，它根据配置文件设置占用逻辑核 cpu0 和控制核 cpu3，第二个 worker 进程 proc_id 为 1，且为 secondary，占用逻辑核 cpu1 和控制核 cpu3。进程 proc_id 依次递增，每个 worker 进程都使用自己分配的逻辑核处理数据业务，控制核所有 worker 进程一起公用，做其它更新配置等业务。

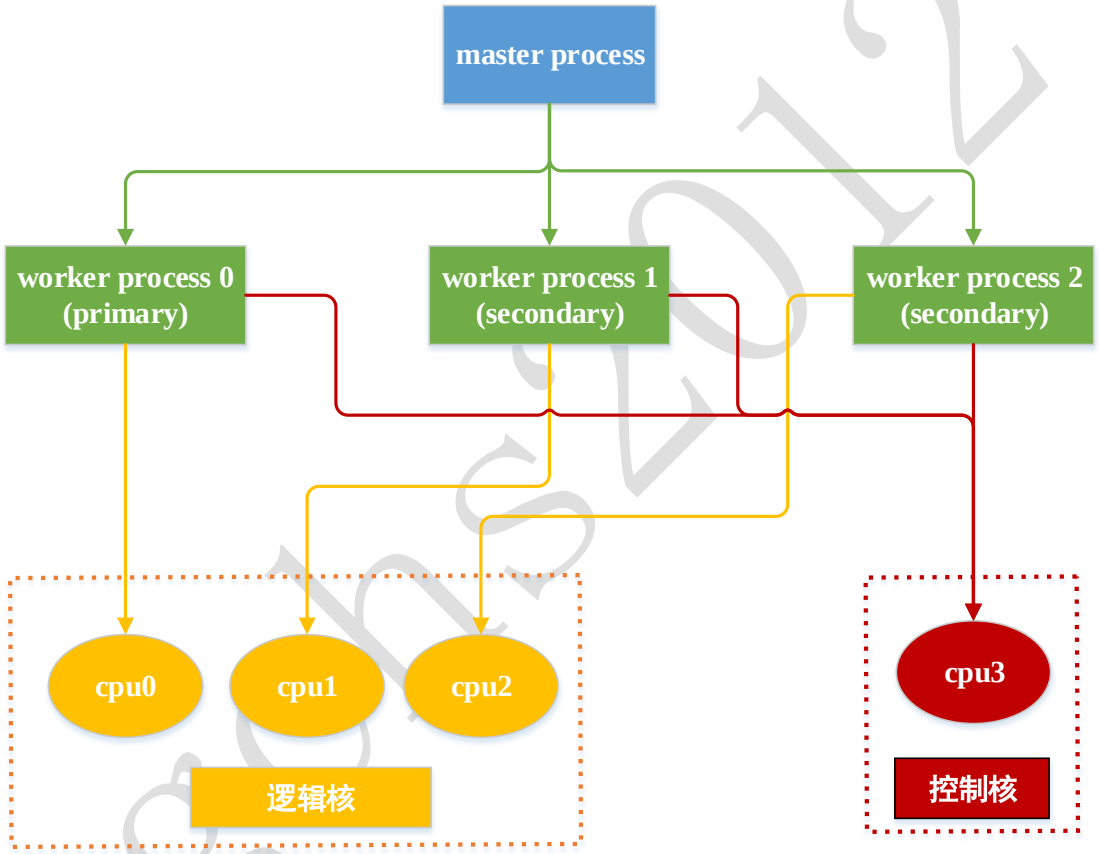

图 5-1 多进程模式

5.2 控制核与逻辑核初始阶段

在逻辑核中通过 IF_CTRL_THRD_OK()判断控制核业务是否初始化完成，如果没有完成就会等待控制核初始化完成后，才进入逻辑核业务处理流程，如图 5-2 所示。

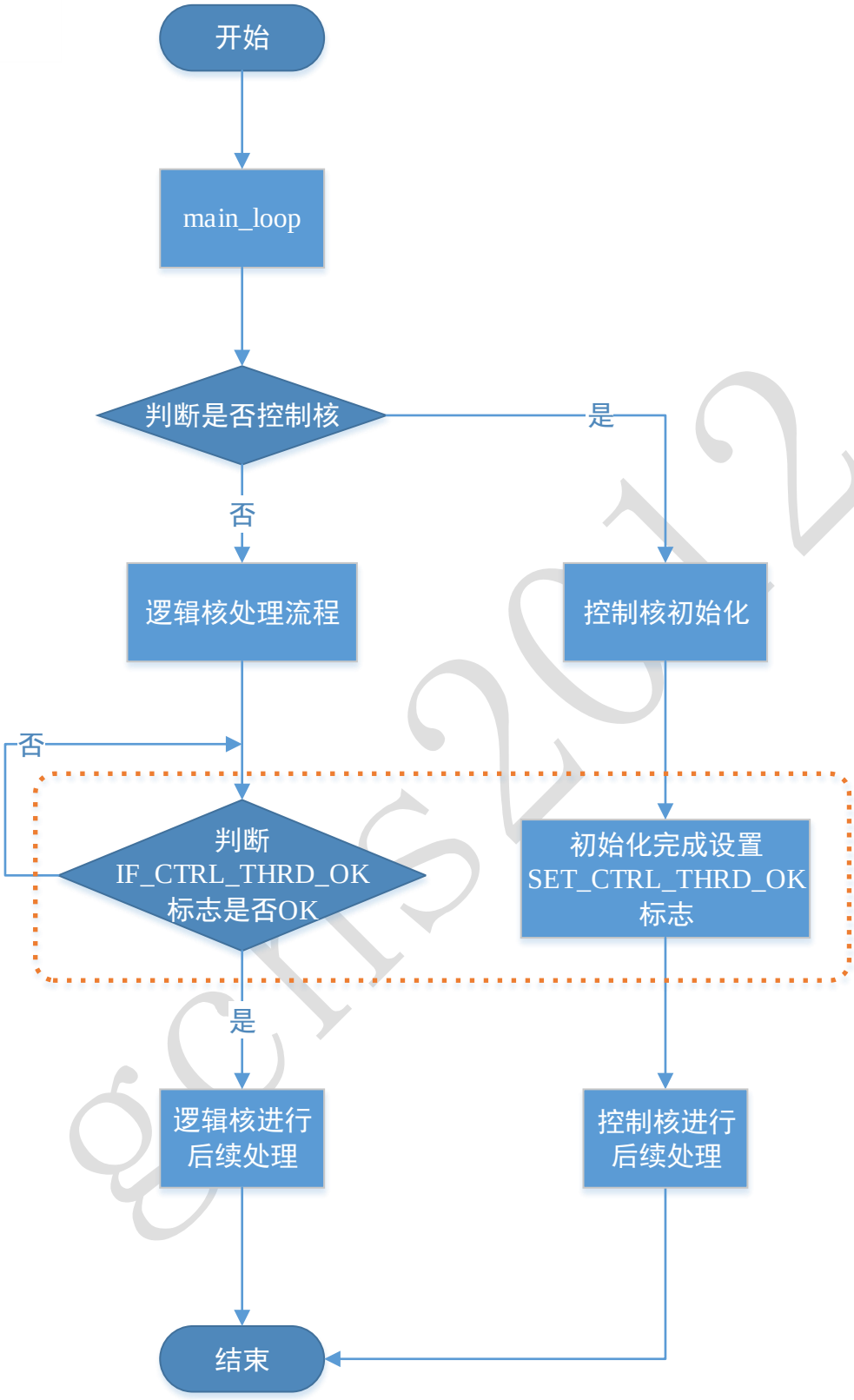

图 5-2 逻辑核与控制核初始阶段

5.3 逻辑核处理流程

图 5-3 展示单次数据包处理流程，dpdk 从网卡接收到数据包后，进入数据处理阶段，通过对数据包进行分析，是否 NM，如果是则根据相关配置，判断数据包是否需要进行克隆保存 ring 中，然后原始数据包将会继续进行后续处理。如果为非 NM，则数据包不会进行克隆保存，将会继续进行后续处理流程。nm_engine_ring 保存克隆的数据包，用于检测引擎从中读取数据包进行处理。

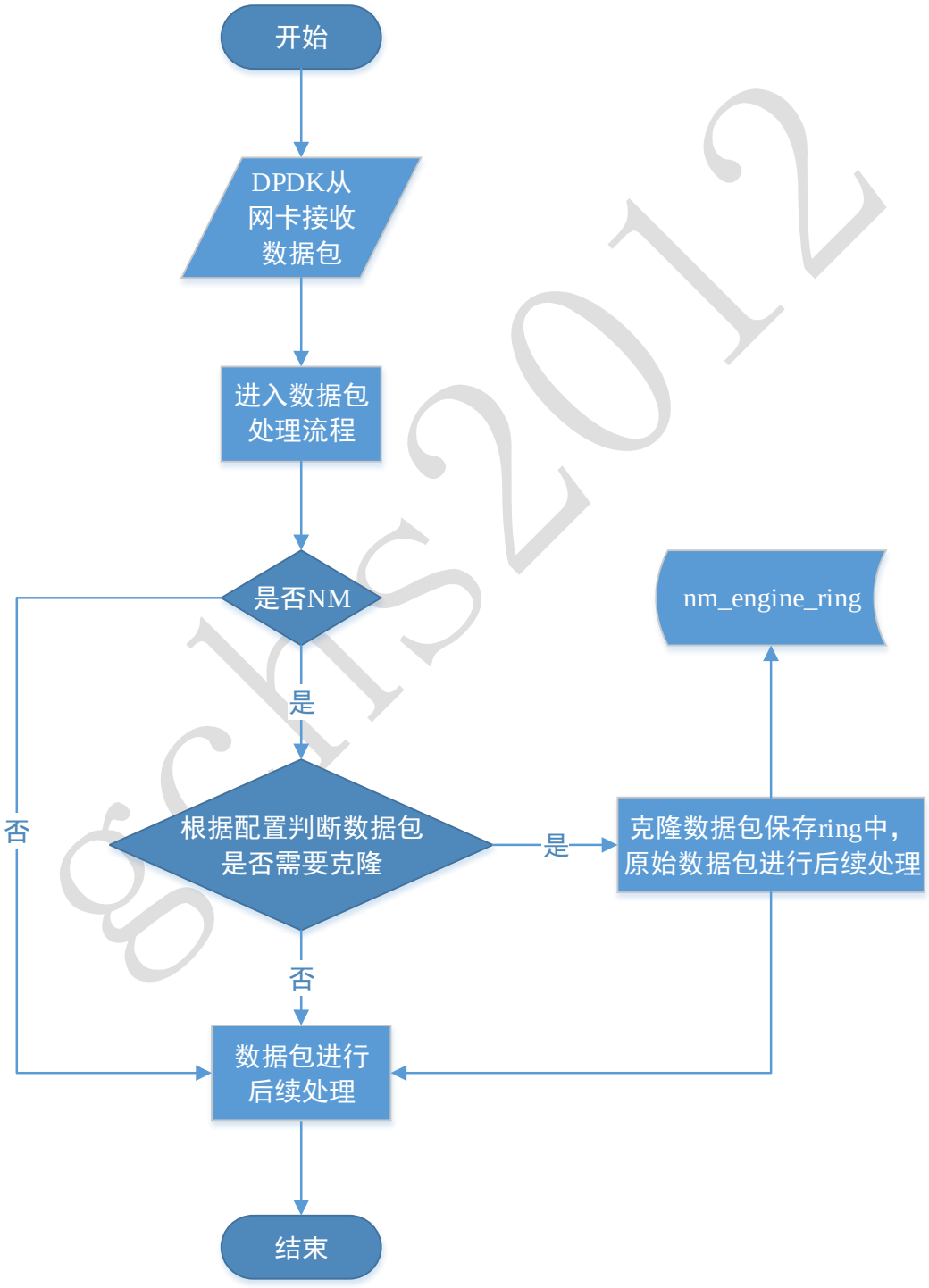

图 5-3 逻辑核数据包处理流程

5.4 控制核处理流程

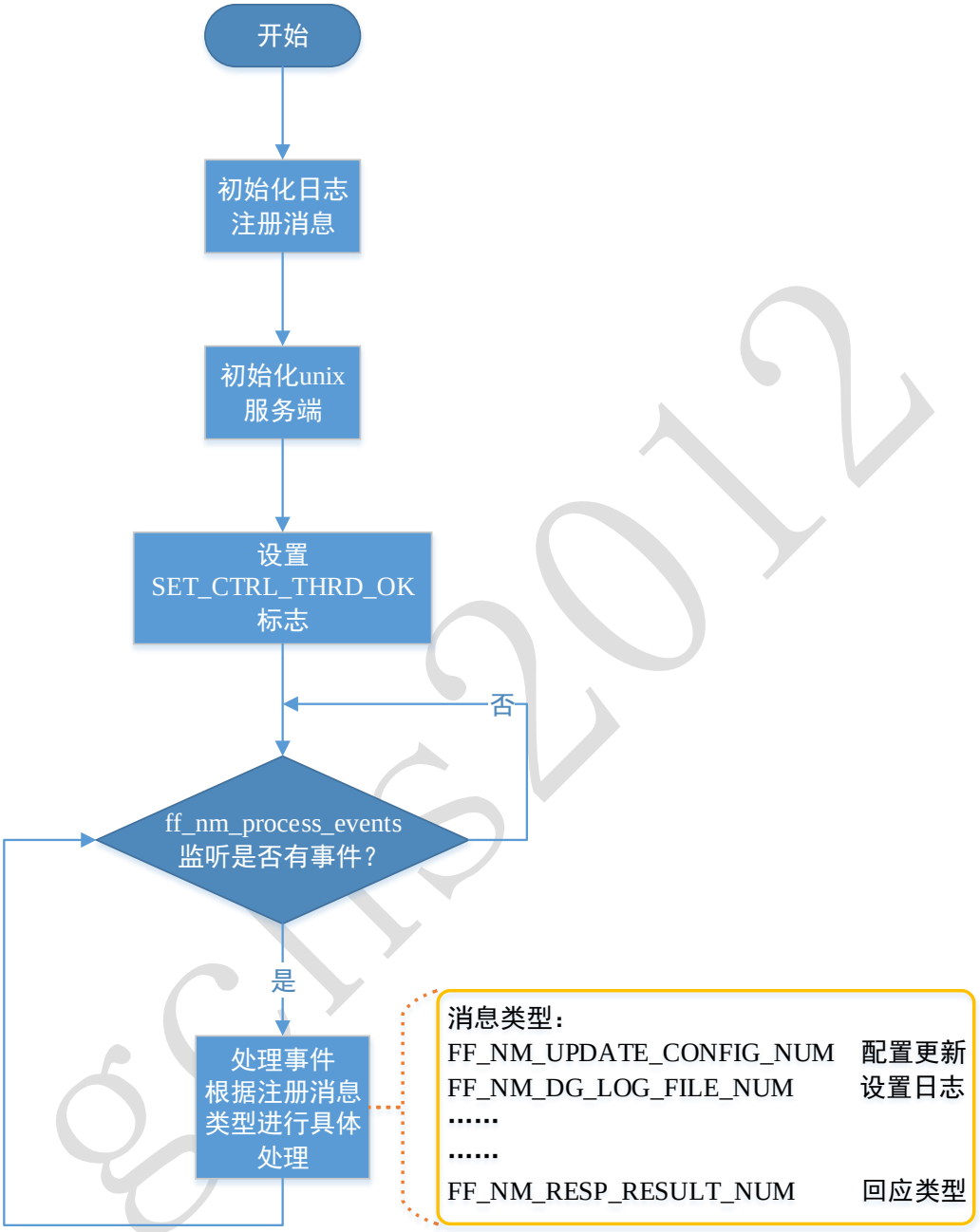

图 5-4 控制核处理流程

6 源码修改

6.1 配置解析

文件 ff_config.c 的 ini_parse_handler 函数添加对控制核解析和逻辑核列表配置支持。

```
static int
ini_parse_handler(void* user, const char* section, const char* name,
        const char* value)
{
        ……..……省略…….....…

        } else if (MATCH("dpdk", "numa_on")) {
                pconfig->dpdk.numa_on = atoi(value);
        } else if (MATCH("dpdk", "tso")) {
                pconfig->dpdk.tso = atoi(value);
        } else if (MATCH("dpdk", "vlan_strip")) {
                pconfig->dpdk.vlan_strip = atoi(value);
        /* BEGIN: Added by zhangcan, 2018/7/26 */
        } else if (MATCH("dpdk", "llcore_list")) {
                return parse_lcore_list(pconfig, value);
        } else if (MATCH("dpdk", "control_core")) {
                return parse_control_core(pconfig, value);
        }
        /* END:      Added by zhangcan, 2018/7/26 */
        else if (MATCH("kni", "enable")) {
                pconfig->kni.enable= atoi(value);
        }
        ……..省略……..…….

        return 1;
}
```

parse_lcore_list 函数用于解析逻辑核列表，parse_control_core 函数用于解析控制核。parse_control_core 函数除解析控制核以外，还设置 dpdk 初始化-c 或-l 参数值，如以下红色部分代码段。

```
static int

parse_control_core(struct ff_config *cfg, const char *value)

{

        uint16_t lcore_id;

        uint16_t ccore_id;

        uint32_t core_mask;

        char buf[128] = {0};


        ccore_id = atoi(value);
```

```
    if (ccore_id == 0) {
        printf("control core is zero core.\n");
        return 0;
    }

    lcore_id = cfg->dpdk.proc_lcore[cfg->dpdk.proc_id];
    cfg->dpdk.control_core = ccore_id;

    if (cfg->dpdk.proc_mask) {
        core_mask  = 1 << ccore_id;
        core_mask += 1 << lcore_id;
        snprintf(buf, sizeof(buf), "%x", core_mask);
        cfg->dpdk.proc_mask = strdup(buf);
    } else if (cfg->dpdk.lcore_list) {
        snprintf(buf, sizeof(buf), "%d,%d",
            cfg->dpdk.proc_lcore_id, ccore_id);
        cfg->dpdk.lcore_list = strdup(buf);
    } else {
        return 0;
    }

    return 1;
}
```

6.2 设置 dpdk 初始化参数

设置 dpdk 初始化参数，添加-l 参数方式的支持，如下红色部分代码段。

```
static int
dpdk_args_setup(struct ff_config *cfg)
{
```

```
    int n = 0, i;
    dpdk_argv[n++] = strdup("f-stack");
    char temp[DPDK_CONFIG_MAXLEN] = {0};

    if (cfg->dpdk.no_huge) {
        dpdk_argv[n++] = strdup("--no-huge");
    }
    if (cfg->dpdk.proc_mask) {
        sprintf(temp, "-c%s", cfg->dpdk.proc_mask);
        dpdk_argv[n++] = strdup(temp);
    }
    /* BEGIN: Added by zhangcan, 2018/8/7 */
    if (cfg->dpdk.lcore_list) {
        sprintf(temp, "-l%s", cfg->dpdk.lcore_list);
        dpdk_argv[n++] = strdup(temp);
    }
    /* END:    Added by zhangcan, 2018/8/7 */
    if (cfg->dpdk.nb_channel) {
        sprintf(temp, "-n%d", cfg->dpdk.nb_channel);
        dpdk_argv[n++] = strdup(temp);
    }
    if (cfg->dpdk.memory) {
        sprintf(temp, "-m%d", cfg->dpdk.memory);
        dpdk_argv[n++] = strdup(temp);
    }
    if (cfg->dpdk.proc_type) {
        sprintf(temp, "--proc-type=%s", cfg->dpdk.proc_type);
        dpdk_argv[n++] = strdup(temp);
    }

    dpdk_argc = n;

    return n;
}
```

6.3 初始化克隆 pktmbuf 内存池和 nm_engine_ring

文件 ff_dpdk_if.c 的 ff_dpdk_init 函数添加 init_nm_engine_ring，主要用于申请克隆数据包 pktmbuf 内存池和申请保存克隆数据包 ring。

```
ff_dpdk_init(int argc, char **argv)
{
        ……..省略……..…….

        init_msg_ring();

        /* BEGIN: Added by zhangcan, 2018/7/30 */
        init_nm_engine_ring();
        /* END:      Added by zhangcan, 2018/7/30 */

        enable_kni = ff_global_cfg.kni.enable;
        if (enable_kni) {
                init_kni();
        }

        ……..省略……..…….

        return 0;
}
```

```
static int
init_nm_engine_ring(void)
{
        char s[128];
        unsigned socketid = 0;
        uint16_t i, lcore_id;

        for (i = 0; i < ff_global_cfg.dpdk.nb_procs; i++) {
                lcore_id = ff_global_cfg.dpdk.proc_lcore[i];
                if (numa_on) {
                        socketid = rte_lcore_to_socket_id(lcore_id);
                }

                if (socketid >= NB_SOCKETS) {
                        rte_exit(EXIT_FAILURE, "Socket %d of lcore %u is out of "
                                "range %d\n", socketid, i, NB_SOCKETS);
                }

                if (nm_engine_pktmbuf_pool[socketid] != NULL) {
                        continue;
                }

                if (rte_eal_process_type() == RTE_PROC_PRIMARY) {
                        snprintf(s, sizeof(s), "nm_engine_mbuf_pool_%d", socketid);
```

```
                nm_engine_pktmbuf_pool[socketid] =
                        rte_pktmbuf_pool_create(s, NM_ENGINE_NB_MBUF,
                                NM_ENGINE_MBUF_CACHE_SIZE, 0,
                                RTE_MBUF_DEFAULT_BUF_SIZE, socketid);
        } else {
                snprintf(s, sizeof(s), "nm_engine_mbuf_pool_%d", socketid);
                nm_engine_pktmbuf_pool[socketid] = rte_mempool_lookup(s);
        }

        if (nm_engine_pktmbuf_pool[socketid] == NULL) {
                rte_exit(EXIT_FAILURE, "Cannot create nm engine mbuf pool "
                        "on socket %d\n", socketid);
        } else {
                printf("create mbuf pool on socket %d\n", socketid);
        }
    }

    /*  创建 NM 引擎 ring */
    nm_engine_ring = create_ring(NM_ENGINE_RING_NAME,
        NM_ENGINE_RING_SIZE, lcore_conf.socket_id, 0);
    if (nm_engine_ring == NULL) {
        rte_panic("create ring::%s failed!\n", NM_ENGINE_RING_NAME);
    }

    return 0;
}
```

6.4 逻辑核和控制核业务处理

ff_dpdk_if.c 文件的 main_loop 为主要业务处理函数，所以数据包读取、处理、发送都在这个函数循环进行。目前在此函数中添加控制业务处理流程，init_control_thread 函数为控制核业务处理，包括后期配置更新等其它业务预留。

```
static int
main_loop(void *arg)
{
    ……..省略……..…….

    qconf = &lcore_conf;

    /* BEGIN: Modified by zhangcan, 2018/7/26 */
    const unsigned lcore_id = rte_lcore_id();
```

```
        if (lcore_id == ff_global_cfg.dpdk.control_core) {
                init_control_thread(lcore_id, qconf->proc_id);
        } else {
                /* wait the control thread ok */
                while (!IF_CTRL_THRD_OK()) usleep(100000);

                printf("--- main_loop: start run lcore(%d).\n", lcore_id);

                while (1) {
                        cur_tsc = rte_rdtsc();
                        if (unlikely(freebsd_clock.expire < cur_tsc)) {
                                rte_timer_manage();
                        }

                        ……..省略……..…….

                                /* Handle remaining prefetched packets */
                                /*  处理剩余的数据包  */
                                for (; j < nb_rx; j++) {
                                        process_packets(port_id, queue_id, &pkts_burst[j], 1, ctx, 0);
                                }
                        }
                }
        }
        /* END:     Modified by zhangcan, 2018/7/26 */

        return 0;
}
```

```
static void
init_control_thread(unsigned lcore_id, uint16_t proc_id)
{
        ff_nm_msg_init(proc_id);
        ff_nm_init_server(proc_id);

        SET_CTRL_THRD_OK();

        printf("--- main_loop: start run control core(%d).\n", lcore_id);

        while (1) {
                ff_nm_process_events();
        }
}
```

6.5 克隆数据包

通过 protocol_filter_duplicate 接口判断，数据包是否 TCP 数据包，后续再添加根据配置判断是否自己监控的 IP 数据包进行指定克隆数据包。

```
static inline void
process_packets(uint16_t port_id, uint16_t queue_id, struct rte_mbuf **bufs,
        uint16_t count, const struct ff_dpdk_if_context *ctx, int pkts_from_ring)
{
            ……..省略……..……

            enum FilterReturn filter = protocol_filter(data, len);
            if (filter == FILTER_ARP) {

            ……..省略……..……

            } else {
                /* BEGIN: Added by zhangcan, 2018/7/25 */
                /*  只需将 TCP 数据包添加到 ring 中  */
                filter = protocol_filter_duplicate(data, len);
                if (filter == FILTER_TCP) {
                    struct rte_mbuf *mbuf_clone;
                    struct rte_mempool *mbuf_pool;

                    /*  克隆数据包  */
                    mbuf_pool = nm_engine_pktmbuf_pool[qconf->socket_id];
                    mbuf_clone = pktmbuf_deep_clone(rtem, mbuf_pool);
                    if (mbuf_clone) {
                        int ret = rte_ring_mp_enqueue(nm_engine_ring, mbuf_clone);
                        if (ret < 0) {
                            rte_pktmbuf_free(mbuf_clone);
                        }
                    }
                }
                /* END:      Added by zhangcan, 2018/7/25 */

                ff_veth_input(ctx, rtem);
            }
        }
}
```

7 测试代码

7.1 配置更新测试客户端

SVN 路径：https://192.190.20.102/svn/DPS/f-stack-1.12/tools/nmconfig

7.2 克隆数据包 ring 测试进程

SVN 路径：https://192.190.20.102/svn/DPS/f-stack-1.12/dpdk/examples/multi_process/simple_sc