作って覚えるDPDKプログラミング

Internet Week 2016

Dec 1, 2016

（株）インターネットイニシアティブ

沖　勝 m-oki@iij.ad.jp

Agenda

- DPDKの概要
- さっそく作ってみる
- どんなふうに動いてるの?
- DPDKが提供している機能の紹介
- DPDKを使った高速化の秘訣
- 動かすには下準備が必要
- いくつかの疑問
- DPDKプログラミングのまとめ

DPDKの概要

ずばり、なんなのか

- **D**ata **P**lane **D**evelopment **K**it
- http://dpdk.org/
- 高速パケットI/O機能を提供するライブラリ。
  - Over 160Mfpsとのこと。10GbEワイヤーレートとか出せます
  - 単体動作するスイッチやルータではありません
  - プロトコルスタックでもありません
- C言語で書かれています。
- Linux, FreeBSDで使えます。
- x86だけでなくARMやPower8, Tile-GXでも動きます。
- ソース提供されているので基本はビルド。
- 最近はパッケージ化されたものもあります。
- BSD Licenseです。
  - Linux kernel moduleはGPLv2

DPDKの歴史

- 2012年9月 version 1.2.3 first public release
  - 32/64bit x86 Linuxのみ
  - NICはIntelのigb(GbE)とixgbe(10GbE)のみ
  - 当時は “Intel DPDK”
- 2016年4月よりバージョン命名規則が 年.月に
- 現時点の最新は 16.11
  - Power8, Tile-GX, ARM(Cavium, RehiveTech, NXP)でも動作
  - Mellanox, Broadcom, Qlogic等のNICにも対応
  - Crypto driver(AES等の暗号化サポート)も提供されている
- 以降 17.02, 17.05, 17.08, 17.11と年4回リリース

さっそく作ってみる

初期化コード

```
#define NPORT 2
#define NRXQ 1
#define NTXQ 1
#define QLEN 144

init(int argc, char *argv[]) {
        rte_eal_init(argc, argv);
        rte_pktmbuf_pool_create("mbufpool", NB_MBUF, cache_size,
                                        0, MBUF_SIZE, rte_socket_id());
        for (portid= 0; portid < NPORT; portid++) {
                rte_eth_dev_configure(portid, NRXQ, NTXQ, &portconf);
                for (n = 0; n < NRXQ; n++) {
                        rte_eth_rx_queue_setup(portid, n, QLEN, …);
                }
                for (n = 0; n < NTXQ; n++) {
                        rte_eth_tx_queue_setup(portid, n, QLEN, …);
                }
        }
}
```

コマンドライン処理

パケット用メモリプールの確保

ポートの初期化

受信キュー初期化

送信キュー初期化

Port0→Port1へパケット転送

```
main(int argc, char *argv[]) {
        init(argc, argv);
        for (;;) {
                npkts = rte_eth_rx_burst(0, mbufs, NB_MBUF);
                rte_eth_tx_burst(1, mbufs, npkts);
        }
}
```

初期化コード呼び出し

永久ループ

Port0からパケット受信

パケットをPort1に送信

起動コマンドライン: sudo fwd_sample -c1 -n2

複数スレッドでパケット処理

```
struct ports {
        int inport;
        int outport;
} ports[] = {
        { 0, 1 },
        { 1, 0 }
};

thread() {
        inport = ports[rte_lcore_id()].inport;
        outport = ports[rte_lcore_id()].outport;

        for (;;) {
                npkts = rte_eth_rx_burst(inport, mbufs, NB_MBUF);

                rte_eth_tx_burst(outport, mbufs, npkts);
        }
}
```

ここでは2スレッド動作を
前提としています

スレッドのエントリ関数

スレッドごとに異なるIDにより
送受信ポートを決定

ひとつのスレッドの処理は
最初のサンプルと同じ

Port0←→Port1パケット転送

```
main(int argc, char *argv[]) {

        init(argc, argv);

        rte_eal_mp_remote_launch(thread, NULL,
                                 CALL_MASTER);

        rte_eal_mp_wait_lcore();
}
```

初期化コード呼び出し

各コアでthread()を実行

全てのthread()の終了待ち

起動コマンドライン: sudo fwd_sample2 -c3 -n2

(16進数)3はbit0,bit1が1
→core0,core1を使用する

メモリチャネル数
1,2,4のいずれか

望んだアプリにする

- 受信と送信の間に処理を挟み込む
- 送信先をダイナミックに決定する
- パケットの中身を加工する
- がんばれば、スイッチやルータも作れます
- プロトコルスタックが不要な処理で威力を発揮
  - OpenFlow
  - トラフィックジェネレータ
  - ロードバランサー
  - など

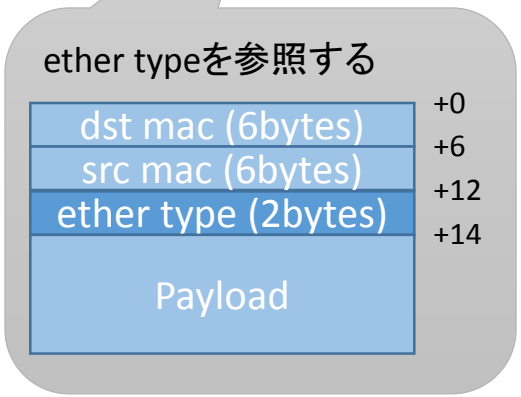
パケットデータの参照・操作

- rte_eth_rx_burst()などで扱うのはmbuf配列
- struct rte_mbuf *
  - *BSD mbufそのものではないが似ている
  - head roomやtail roomをあらかじめ空けてある
- 主なAPI
  - Rte_pktmbuf_alloc()　mbufの確保（bulk版APIもある）
  - rte_pktmbuf_free()　mbufの解放
  - rte_pktmbuf_mtod()　mbuf先頭データの取り出し
  - rte_pktmbuf_len()　パケット長の取得
  - rte_pktmbuf_prepend()　先頭に指定バイト数加える
  - rte_pktmbuf_adj()　先頭から指定バイト数取り除く
  - Rte_pktmbuf_append()　末尾に指定バイト数加える
  - rte_pktmbuf_trim()　末尾から指定バイト数取り除く

パケットデータを扱う例

```
struct rte_mbuf *mbufs[NB_MBUF];

n_mbufs = rte_eth_rx_burst(portid, mbufs, NB_MBUF);

for (n = 0; n < n_mbufs; n++) {
        mbuf = mbufs[n];

        typeoff = rte_pktmbuf_mtod_offset(mbuf, uint16_t *, 12);


        switch (*typeoff) {
        case ETHERTYPE_IP:
                my_ip_input(mbuf);
                break;
        default:
                rte_pktmbuf_free(mbuf);
        }
}
```

先頭からのオフセット指定し、
指定の型で取り出す



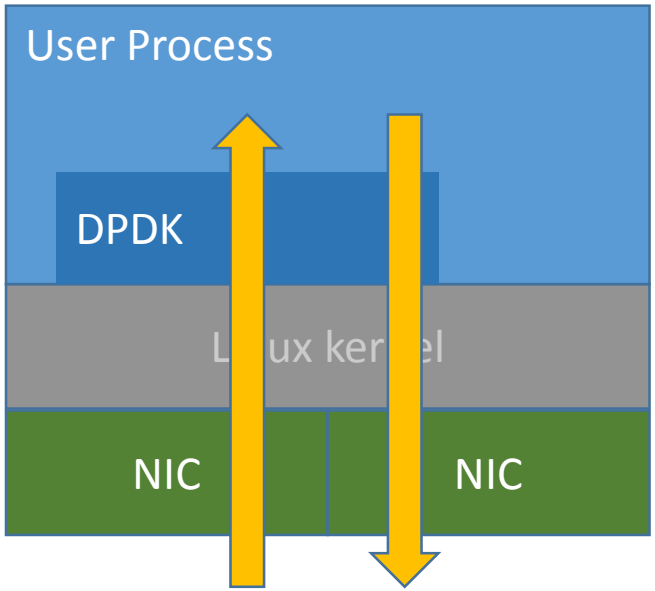
ここで当然の疑問

- 1スレッド1コア? → そのとおりです
- 永久ループ、つまり、ぐるぐるまわる
- 受信APIはブロックするか? **→しません**
  - パケットを受信していなければ0個
- pollやselect相当のAPIは? **→ありません**
- CPU利用率? **→ 100%**
- 速度至上主義。
  CPU loadどれだけ食おうが、とにかく速く
- コアやCPUソケットをめちゃくちゃ意識します

どんなふうに動いてるの?

おおまかな動作イメージ

- kernelをバイパスしてDPDKでパケット受信
- パケット送信もDPDKによってkernelをバイパス
- 動作の主体はUser Process



PMD (Poll Mode Driver)

- Linuxのuio(userspace I/O) kernel moduleを利用
- DPDKでビルドされるigb_uio.koを組み込み
- NICのドライバはすべてDPDKの中で実装
- 割込みを使わずポーリングするドライバのため PMD (Poll Mode Driver) と名付けられた
- 最初の実装はFreeBSDのドライバからの移植

コンテキストスイッチの抑制

- スレッドが走るコアを固定する
  - pthread_setaffinity_np(3)
  - 1スレッド1コア。4スレッドなら4コア必要。
- PMDによって送受信割り込みを抑制
- Page faultを抑制するためのhugepageの利用
  - 通常4KB/pageで実メモリがないとPage faultで確保
  - TLB miss例外が発生して確保後に元の処理に戻る
  - hugepageは2MB/pageあるいは1GB/page
  - データベースの高速化にも使われる

ゼロコピー

- パケット処理の際にコピーを不要とする。
- 受信時にhugepageにパケットデータを書き込みポインタをユーザプログラムに渡す。
- ユーザプログラムがポインタを送信APIに渡す。
- 送信APIはコピーせずそのまま送信処理を実行。

DPDKが提供している機能

DPDKの主なライブラリの紹介

- Environment Abstraction Layer (librte_eal)
- Ethernet関連 (librte_ether)
- パケットデータ操作 (librte_mbuf)
- LPM (librte_lpm)
- ACL (librte_acl)
- ハッシュ関数、ハッシュテーブル (librte_hash)
- パケットのリオーダリング (librte_reorder)
- IPフラグメント処理 (librte_ip_frag)
- Locklessなリングバッファ (librte_ring)
- など。
- ドキュメントにて解説されています（英語ですが）
  - http://dpdk.org/doc/guides/prog_guide/

サンプルプログラム

```
~src/dpdk$ ls examples/
bond                ipv4_multicast   link_status_interrupt  quota_watermark
cmdline             kni              load_balancer          rxtx_callbacks
distributor         l2fwd            Makefile               skeleton
dpdk_qat            l2fwd-cat        multi_process          tep_termination
ethtool             l2fwd-crypto     netmap_compat          timer
exception_path      l2fwd-jobstats   nohuge-test            vhost
helloworld          l2fwd-keepalive  packet_ordering        vhost_xen
ip_fragmentation    l3fwd            performance-thread     vmdq
ip_pipeline         l3fwd-acl        ptpclient              vmdq_dcb
ip_reassembly       l3fwd-power      qos_meter              vm_power_manager
ipsec-secgw         l3fwd-vf         qos_sched
~/src/dpdk$
```

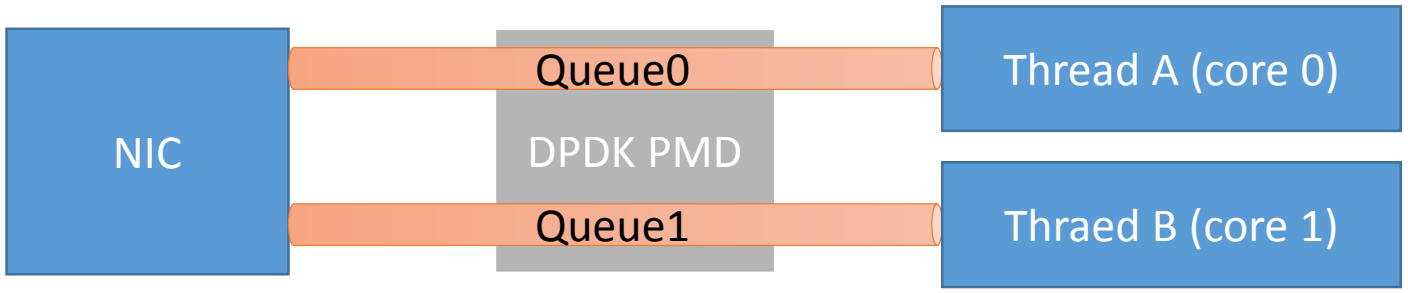
DPDKを使った高速化の秘訣

DPDKを使った高速化の秘訣

- ハードウェアリソースを活用する
  - NICのオフロード機能 (checksum, TSO)
  - マルチキューNIC (RSS, flow director)
- 可能ならスレッド間でリソースを共有しない。
  - たとえばスレッド（コア）ごとに持たせる
  - マルチキューNICのキューごとにコアを割り当てる
- CPUがなるべく待たないようにする。
  - なるべくロックしない
- なるべくパケットをバルクで処理する。
- なるべくコピーしない。

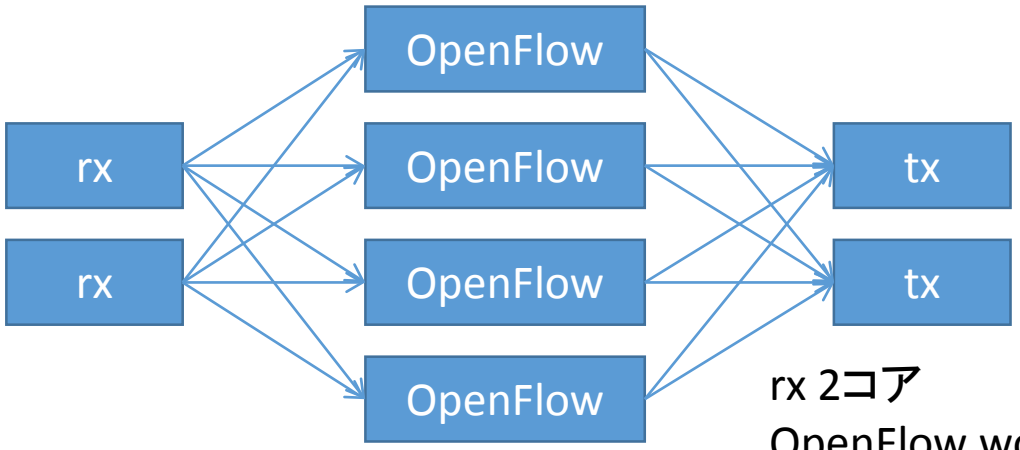
ハードウェア活用例

- マルチキューNICを利用
  - IntelのGbE NICでは最大8 queue
- RSS(Receive Side Scaling; NICの機能)で振り分け
  - IPv4 src, dstの組からhash値を計算し振り分ける
  - 送信先におけるパケット順序性が保証される
- それぞれのスレッドが互いを気にせず処理



高速化の秘訣2

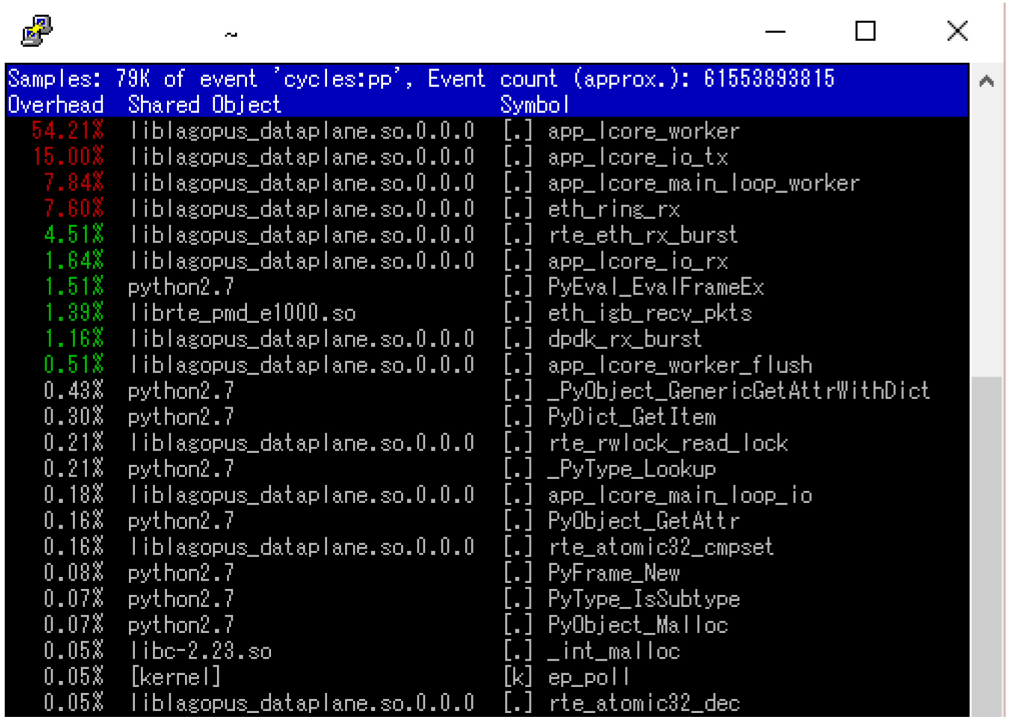
- コアごとに処理内容を分ける
  - たとえばI/O処理とパケットフィルタリング
  - OSSのOpenFlowスイッチLagopusの手法



rx 2コア
OpenFlow worker 4コア
tx 2コア

ボトルネックの調査

- perfコマンド
  - サブコマンドがいろいろあるがまずはperf top
  - 空ループも高負荷に見える点に注意



動かすには下準備が必要

下準備: 実は大きなハードル

- DPDKのビルド
  - DPDKのトップディレクトリ$RTE_SDKにcdしておき **`./tools/setup.sh`** を実行。対話形式でビルドできる。
  - 対話形式でなくmakeを使うときは下記のようにする。
    - **`make T=x86_64-native-linuxapp-gcc config`**
    - **`make`**
- hugepageの予約
  - Linux kernel起動パラメータに追加
  - Ubuntuなら/etc/default/grubを編集してupdate-grub
    - 例: **`GRUB_CMDLINE_LINUX="hugepages=2048"`**
  - /etc/fstabにエントリ追加
    - **`none /mnt/huge hugetlbfs defaults 0 0`**
  - 一度再起動が必要

もう一つの下準備

- PMD動作に必要なカーネルモジュール組み込み
  - **`sudo modprobe uio`**
  - **`sudo insmod $RTE_SDK/build/kmod/igb_uio.ko`**
- UIOを使うようNICのドライバの差し替え
  - **`sudo $RTE_SDK/tools/dpdk-devbind.py`**
  - パラメータなしでUsageが表示される
    - 例: **`dpdk-devbind.py --bind=igb_uio 01:00.0`**
    - この例の`01:00.0`はPCIアドレス
    - 値は **`ethtool -i eth1`** など実行するとわかる
  - DPDKのバージョンが古いとスクリプト名が違う
- OSからNICが見えなくなる(!)
- 再起動のたびに上記を実行する必要あり

いくつかの疑問

いくつかの疑問

- 仮想環境でも動く? オーバーヘッドは?
  - 動作します。virtio PMDやvhost PMDを使えます。
  - SR-IOVも使えます。
  - CPU100%については後述
- 通常OSの処理よりいいって本当?
  - DPDKはプロトコルスタックを持っていません。
  - DPDKやthird partyソフトウェアがない機能は自作が必要。
  - 最近カーネル内で完結するフレームワークが話題だが（eBPF, XDP）、一長一短。
- 一部のパケットだけDPDKで処理したい。可能?
  - 可能と言えば可能。
  - DPDK提供のKNI(Kernel Network Interface)かtapを使う。
  - OSで処理させるパケットのスループットは落ちる。
  - うまくすみわけできそうな例
    - L3処理で速度を要求されないICMPはカーネルに、その他はDPDKで

CPU100%問題解決の糸口

- DPDKのパケット受信に機能が追加されてます
  - interrupt mode
  - 内部的には、パケット受信割り込みをuioのfdへのpoll/selectで検知し、callback functionを呼び出す
  - DPDKの使い方としては関数を登録してフラグを立てておけばこのモードになる
- Interrupt modeと従来のポーリングループを併用してCPU loadを下げつつ高速転送を実装できそうです
- いわばLinux NAPIのDPDK版
- ただしinterrupt modeがあるだけなので自作が必要

C言語以外で使えますか?

- C++: もちろん使えます（extern ”C” ）
- 他の言語はwrapperを使って呼び出す
  - Go: go-dpdk https://github.com/melvinw/go-dpdk
  - Rust: rust-dpdk https://github.com/flier/rust-dpdk
  - DPDKのを活かす高速性が維持できてるかは不明

DPDK関連のOSSを少し紹介

- Pktgen-DPDK
  - http://dpdk.org/browse/apps/pktgen-dpdk/refs/
  - DPDKを使ったトラフィックジェネレータ
- Lagopus
  - https://lagopus.github.io
  - OpenFlow 1.3対応ソフトウェアスイッチ
- Seastar
  - http://www.seastar-project.org/
  - サーバーアプリケーション向けのフレームワーク
- VPP
  - https://wiki.fd.io/view/VPP
  - Ciscoのパケット処理フレームワーク
- mTCP
  - https://github.com/eunyoung14/mtcp
  - マルチコアを活用したユーザスペースTCP実装

DPDKプログラミングのまとめ

まとめ

- 導入、下準備は少々面倒。
- (3rd party含め)ライブラリにない機能は全部自分で組む必要がある。
- プログラミング自体は比較的シンプル
- サンプルプログラムも豊富、OSSの活用例も

この機会にDPDKプログラミングを
始めてみませんか?