KLab

第5回 KLab Expert Camp

TCP/IPプロトコルスタック自作開発

Day3

KLab株式会社

本日の目標とタスク

EthernetとARPを実装してホストと通信できるようになる

- Ethernetデバイス
- ARP要求への応答
- ARPキャッシュ
- ARP要求の送信
- ARPタイマー

IPルーティングを実装してインターネット上のホストと通信できるようになる

- IPルーティングの実装

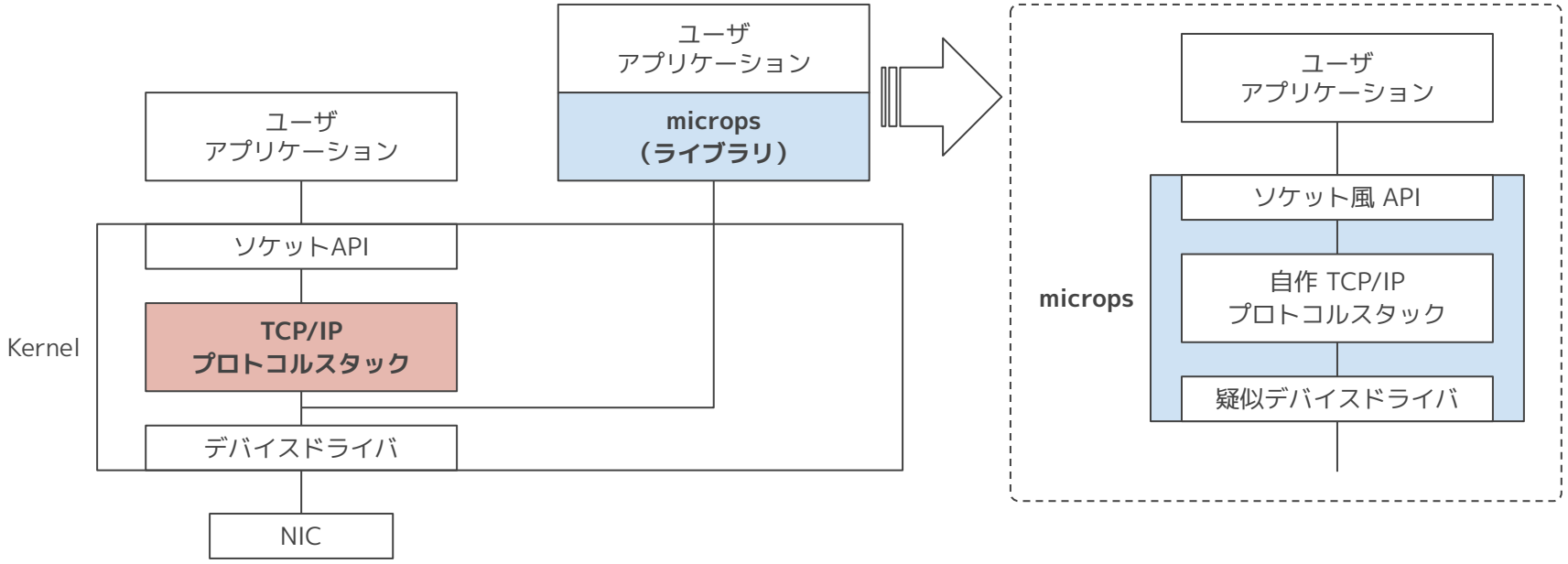
KLab

STEP 12 Ethernetデバイス

microps

ユーザアプリケーション用のライブラリとして実装

- OSのプロトコルスタックを通さずにパケットを処理する



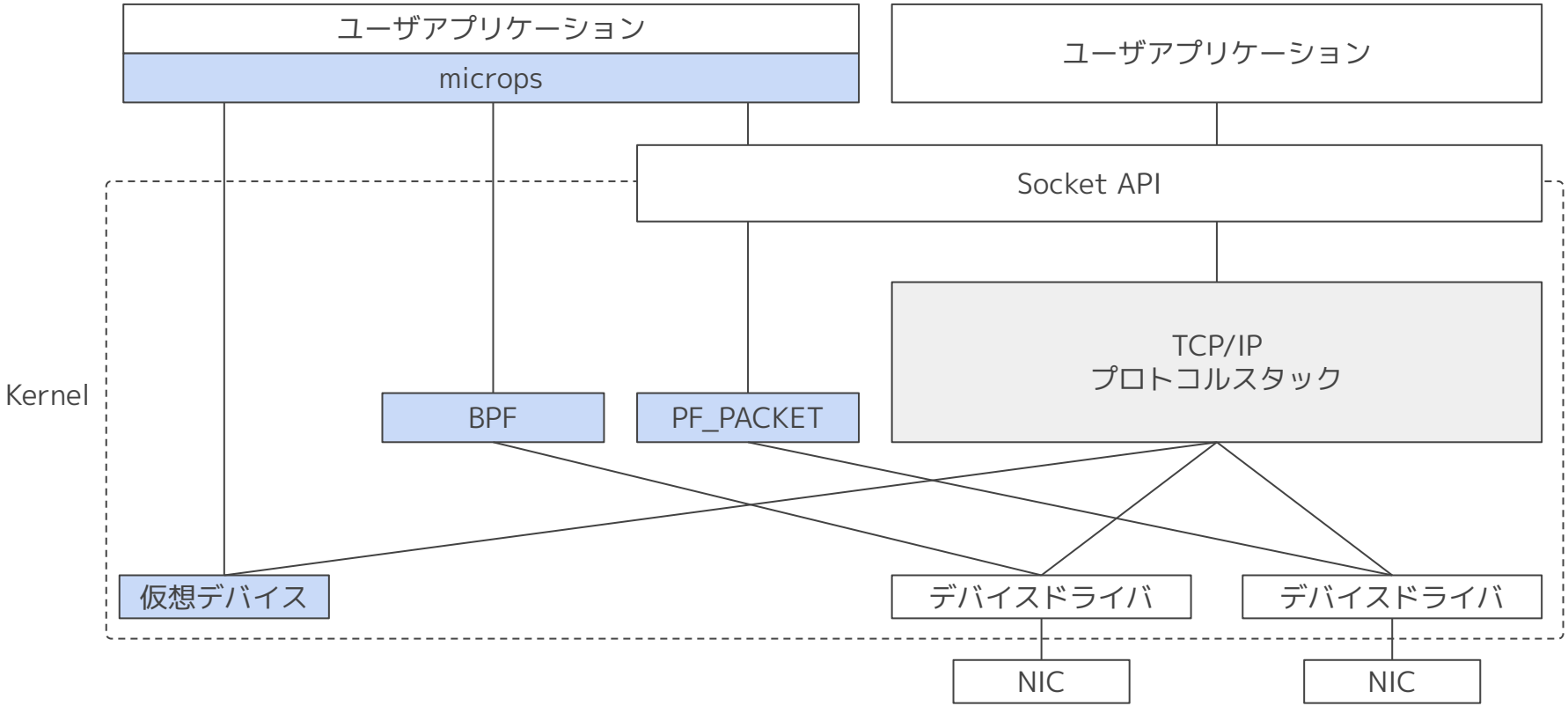
Ethernetフレームを直接送受信する方法

KLab

OS毎に何かしらの手段が提供されている（バラバラで統一されていない）

- Linux
  - Socket (with PF_PACKET) / DPDK / XDP
- BSD
  - BPF - Berkeley Packet Filter
- UNIX (SUSv2)
  - DLPI - Data Link Provider Interface
- Windows
  - NDIS - Network Driver Interface Specification

Ethernetフレームを直接送受信する方法



仮想ネットワークデバイス

KLab

- TUN/TAP（マルチプラットフォーム）
  - TAP デバイスは Ethernet デバイスをエミュレートする
  - write => Ethernetフレームを OS のプロトコルスタックへ Inject
  - read => OS のプロトコルスタックが書き込んだフレームを読み出す
- veth（Linux）
  - 仮想的な Ethernet デバイスのペアを生成
  - 一方に書き込んだフレームがもう一方から読み出される
  - 仮想マシンとホスト間の通信に利用されている

PF_PACKET

- ソケットを通じてリンク層へアクセスする

```
soc = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```

SOCK_RAW: リンク層のヘッダはユーザが処理
SOCK_DGRAM: リンク層のヘッダはドライバが処理

- リンクレベルI/Oのためのプロトコルファミリ
  - Linux 2.4 以降で利用可能
  - ソケット経由でEthernetフレームをダイレクトに送受信

https://gist.github.com/pandax381/31888921e90ee60699867e0ed328e1b0

packet(7) - https://man7.org/linux/man-pages/man7/packet.7.html

BPF

- BPFデバイスを通じてリンク層へアクセスする

```
fd = open("/dev/bpf0", O_RDWR, 0);
```

- BSD（macOSを含む）に実装されているパケットフィルタ機構
  - Linux の eBPF (extend BPF) とは別物
    - 区別するために cBPF (classic BPF) と表記することもある
  - BPFデバイス経由でEthernetフレームをダイレクトに送受信

https://gist.github.com/pandax381/1291f1a1fc636b1c242ff2bc0824f082

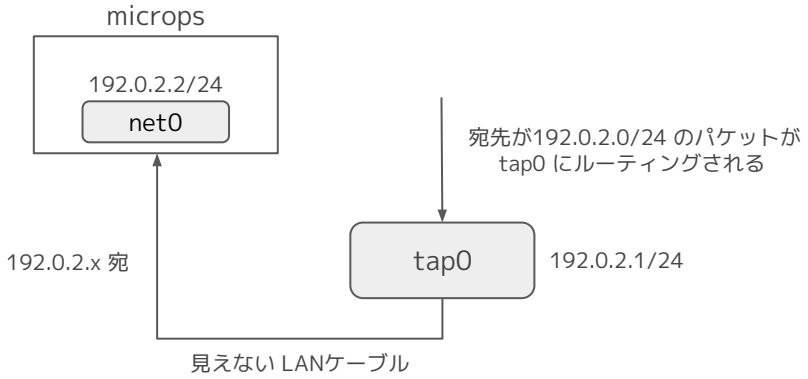
The BSD Packet Filter - https://www.tcpdump.org/papers/bpf-usenix93.pdf
bpf(4) - https://www.freebsd.org/cgi/man.cgi?bpf(4)

TUN/TAP

- プラットフォーム毎に初期化方法が微妙に異なる（今回はLinuxの場合の解説）

```
fd = open("/dev/net/tun", O_RDWR, 0);
```

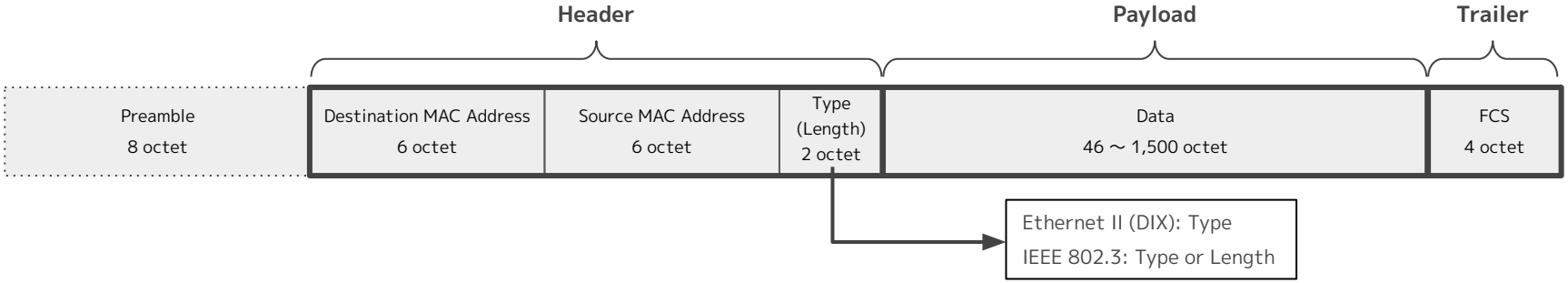
- TUN … L3 (Ethernetヘッダは付かない)
- TAP … L2（Ethernetヘッダが付く）

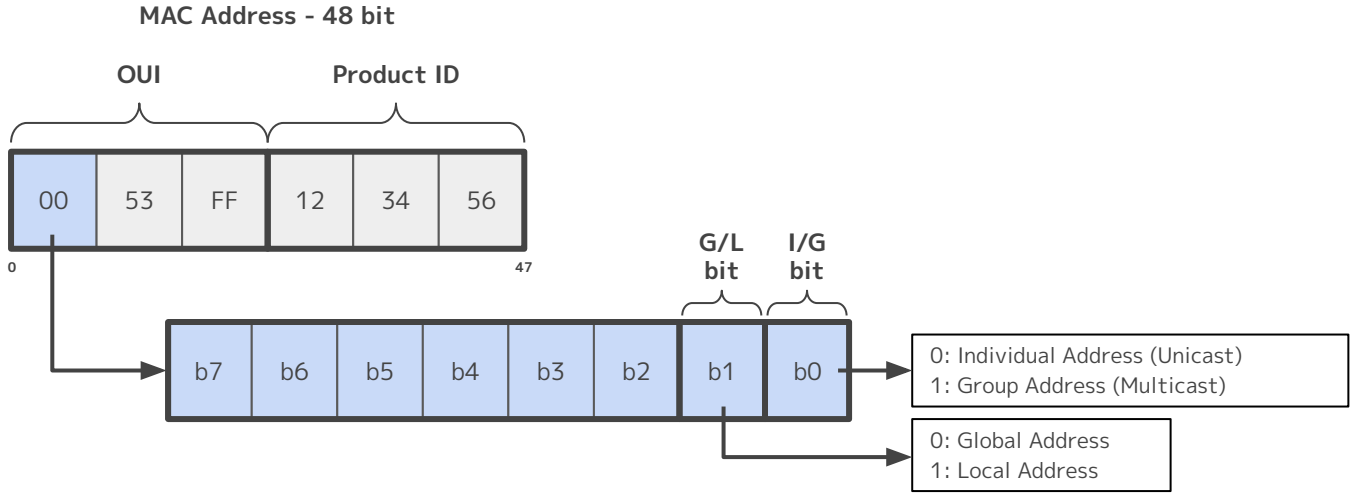


https://gist.github.com/pandax381/de6e20b5dec7eb37ff181a1be1dfef89

Universal TUN/TAP device drive - https://www.kernel.org/doc/Documentation/networking/tuntap.txt
tun(4) - https://www.freebsd.org/cgi/man.cgi?query=tun&sektion=4

Ethernetフレームの構造





このステップの作業

- Ethernetヘッダ構造体
- Ethernetフレームの生成と出力
- Ethernetフレームの入力と検証
- Ethernetデバイス（TAP）のドライバ実装

このステップのコードの雛形

```
> git show c1f511e
```
直近のステップからの差分が表示される

- driver/ether_tap.h 新規
- ether.c 新規
- ether.h 新規
- platform/linux/driver/ether_tap.c 新規

Ethernetヘッダの構造体＆デバッグ出力

KLab

ether.h

```
#define ETHER_ADDR_LEN 6
```

Ethernetアドレス（MACアドレス）のバイト列の長さ

ether.c

```
struct ether_hdr {
    uint8_t dst[ETHER_ADDR_LEN];
    uint8_t src[ETHER_ADDR_LEN];
    uint16_t type;
};

static void
ether_dump(const uint8_t *frame, size_t flen)
{
    struct ether_hdr *hdr;
    char addr[ETHER_ADDR_STR_LEN];

    hdr = (struct ether_hdr *)frame;
    flockfile(stderr);
    fprintf(stderr, "        src: %s\n", ether_addr_ntop(hdr->src, addr, sizeof(addr)));
    fprintf(stderr, "        dst: %s\n", ether_addr_ntop(hdr->dst, addr, sizeof(addr)));
    fprintf(stderr, "       type: 0x%04x\n", ntoh16(hdr->type));
#ifdef HEXDUMP
    hexdump(stderr, frame, flen);
#endif
    funlockfile(stderr);
}
```

Ethernetヘッダの構造体

バイト列のMACアドレスを文字列に変換

Ethernetフレームの生成と出力

ether.h

```
typedef ssize_t (*ether_transmit_func_t)(struct net_device *dev, const uint8_t *data, size_t len);
```

出力用コールバック関数のプロトタイプ宣言に別名（ether_transmit_func_t）をつける

ether.c

```
int
ether_transmit_helper(struct net_device *dev, uint16_t type, const uint8_t *data, size_t len, const void *dst, ether_transmit_func_t callback)
{
    uint8_t frame[ETHER_FRAME_SIZE_MAX] = {};
    struct ether_hdr *hdr;
    size_t flen, pad = 0;

    hdr = (struct ether_hdr *)frame;
    memcpy(hdr->dst, dst, ETHER_ADDR_LEN);
    memcpy(hdr->src, dev->addr, ETHER_ADDR_LEN);
    hdr->type = hton16(type);
    memcpy(hdr+1, data, len);
    if (len < ETHER_PAYLOAD_SIZE_MIN) {
        pad = ETHER_PAYLOAD_SIZE_MIN - len;
    }
    flen = sizeof(*hdr) + len + pad;
    debugf("dev=%s, type=0x%04x, len=%zu", dev->name, type, flen);
    ether_dump(frame, flen);
    return callback(dev, frame, flen) == (ssize_t)flen ? 0 : -1;
}
```

Ethernetフレームの生成
・ヘッダの各フィールドに値を設定
・ヘッダの直後にデータをコピー

最小サイズに満たない場合はパディングを挿入してサイズを調整

引数で渡された関数をコールバックして生成したEthernetフレームを出力する
※ 実際の書込み処理は ether_input_helper() を呼び出したドライバの関数の中で行われる

Ethernetではフレームの最大サイズに加えて最小サイズも規定されており、最小サイズに満たない場合にはパディングを挿入してフレームサイズを調整して送信する。
（CSMA/CDにおいて、フレームサイズが小さすぎるとコリジョンを検出した際のジャム信号が届く前に送信を終えてしまい、衝突を検知できなくなてしまうため）

Ethernetフレームの入力と検証

KLab

ether.h

```
typedef ssize_t (*ether_input_func_t)(struct net_device *dev, uint8_t *buf, size_t size);
```

入力用コールバック関数のプロトタイプ宣言に別名（ether_transmit_func_t）をつける

ether.c

```
int
ether_input_helper(struct net_device *dev, ether_input_func_t callback)
{
    uint8_t frame[ETHER_FRAME_SIZE_MAX];
    ssize_t flen;
    struct ether_hdr *hdr;
    uint16_t type;

    flen = callback(dev, frame, sizeof(frame));
    if (flen < (ssize_t)sizeof(*hdr)) {
        errorf("too short");
        return -1;
    }
    hdr = (struct ether_hdr *)frame;
    if (memcmp(dev->addr, hdr->dst, ETHER_ADDR_LEN) != 0) {
        if (memcmp(ETHER_ADDR_BROADCAST, hdr->dst, ETHER_ADDR_LEN) != 0) {
            /* for other host */
            return -1;
        }
    }
    type = ntoh16(hdr->type);
    debugf("dev=%s, type=0x%04x, len=%zd", dev->name, type, flen);
    ether_dump(frame, flen);
    return net_input_handler(type, (uint8_t *)(hdr+1), flen - sizeof(*hdr), dev);
}
```

引数で渡された関数をコールバックしてEthernetフレームを読み込む
※ 実際の読み込み処理は ether_input_helper() を呼び出したドライバの関数の中で行われ、ether_input_helper() は結果だけ受け取る

読み込んだフレームのサイズがEthernetヘッダより小さかったらエラーとする

Ethernetフレームのフィルタリング
・宛先がデバイス自身のMACアドレスまたはブロードキャストMACアドレスであればOK
・それ以外は他のホスト宛とみなして黙って破棄する

net_input_handler() を呼び出してプロトコルスタックにペイロードを渡す

Ethernetデバイスの共通設定

ether.c

```
void
ether_setup_helper(struct net_device *dev)
{
    dev->type = NET_DEVICE_TYPE_ETHERNET;
    dev->mtu = ETHER_PAYLOAD_SIZE_MAX;
    dev->flags = (NET_DEVICE_FLAG_BROADCAST | NET_DEVICE_FLAG_NEED_ARP);
    dev->hlen = ETHER_HDR_SIZE;
    dev->alen = ETHER_ADDR_LEN;
    memcpy(dev->broadcast, ETHER_ADDR_BROADCAST, ETHER_ADDR_LEN);
}
```

Ethernetデバイス共通のパラメータ

Ethernetデバイス（TAP）オープン/クローズ

platform/linux/driver/ether_tap.c

```
static int
ether_tap_open(struct net_device *dev)
{
    struct ether_tap *tap;
    struct ifreq ifr = {};

    tap = PRIV(dev);
    tap->fd = open(CLONE_DEVICE, O_RDWR);
    if (tap->fd == -1) {
        errorf("open: %s, dev=%s", strerror(errno), dev->name);
        return -1;
    }
    strncpy(ifr.ifr_name, tap->name, sizeof(ifr.ifr_name)-1);
    ifr.ifr_flags = IFF_TAP | IFF_NO_PI;
    if (ioctl(tap->fd, TUNSETIFF, &ifr) == -1) {
        errorf("ioctl [TUNSETIFF]: %s, dev=%s", strerror(errno), dev->name);
        close(tap->fd);
        return -1;
    }

    if (memcmp(dev->addr, ETHER_ADDR_ANY, ETHER_ADDR_LEN) == 0) {
        if (ether_tap_addr(dev) == -1) {
            errorf("ether_tap_addr() failure, dev=%s", dev->name);
            close(tap->fd);
            return -1;
        }
    }
    return 0;
};
```

ioctl() で使うリクエスト/レスポンス兼用の構造体

TUN/TAPの制御用デバイスをオープン

TAPデバイスの名前を設定

フラグ設定（IFF_TAP: TAPモード、IFF_NO_PI: パケット情報ヘッダを付けない）

TAPデバイスの登録を要求

シグナル駆動I/Oのための設定

HWアドレスが明示的に設定されていなかったら

OS側から見えているTAPデバイスのHWアドレスを取得して使用する

```
static int
ether_tap_close(struct net_device *dev)
{
    close(PRIV(dev)->fd);
    return 0;
}
```

ディスクリプタをクローズ

```
/* Set Asynchronous I/O signal delivery destination */
if (fcntl(tap->fd, F_SETOWN, getpid()) == -1) {
    errorf("fcntl(F_SETOWN): %s, dev=%s", strerror(errno), dev->name);
    close(tap->fd);
    return -1;
}
/* Enable Asynchronous I/O */
if (fcntl(tap->fd, F_SETFL, O_ASYNC) == -1) {
    errorf("fcntl(F_SETFL): %s, dev=%s", strerror(errno), dev->name);
    close(tap->fd);
    return -1;
}
/* Use other signal instead of SIGIO */
if (fcntl(tap->fd, F_SETSIG, tap->irq) == -1) {
    errorf("fcntl(F_SETSIG): %s, dev=%s", strerror(errno), dev->name);
    close(tap->fd);
    return -1;
}
```

シグナルの配送先を設定

シグナル駆動I/Oを有効にする

送信するシグナルを指定

シグナル駆動I/O … データが入力可能な状態になったらシグナルを発生させて知らせてくれる

ioctl() でTAPデバイスの生成に成功したら dev->fd を read() / write() することでEthernetフレームの送受信が出来るようになる

Ethernetデバイス（TAP）HWアドレスの取得

platform/linux/driver/ether_tap.c

```
static int
ether_tap_addr(struct net_device *dev)
{
    int soc;
    struct ifreq ifr = {};   ioctl() で使うリクエスト/レスポンス兼用の構造体

    soc = socket(AF_INET, SOCK_DGRAM, 0);
    if (soc == -1) {
        errorf("socket: %s, dev=%s", strerror(errno), dev->name);
        return -1;
    }                                                    なんでもいいのでソケットをオープンする
    strncpy(ifr.ifr_name, PRIV(dev)->name, sizeof(ifr.ifr_name)-1);   ハードウェアアドレスを取得したいデバイスの名前を設定する
    if (ioctl(soc, SIOCGIFHWADDR, &ifr) == -1) {
        errorf("ioctl [SIOCGIFHWADDR]: %s, dev=%s", strerror(errno), dev->name);
        close(soc);
        return -1;
    }                                                    ハードウェアアドレスの取得を要求する
    memcpy(dev->addr, ifr.ifr_hwaddr.sa_data, ETHER_ADDR_LEN);   取得したアドレスをデバイス構造体へコピー
    close(soc);   使い終わったソケットをクローズ
    return 0;
}
```

通信するわけでもないのにソケットをオープンしている理由 … ioctl() の SIOCGIFHWADDR 要求がソケットとして開かれたディスクリプタでのみ有効なため

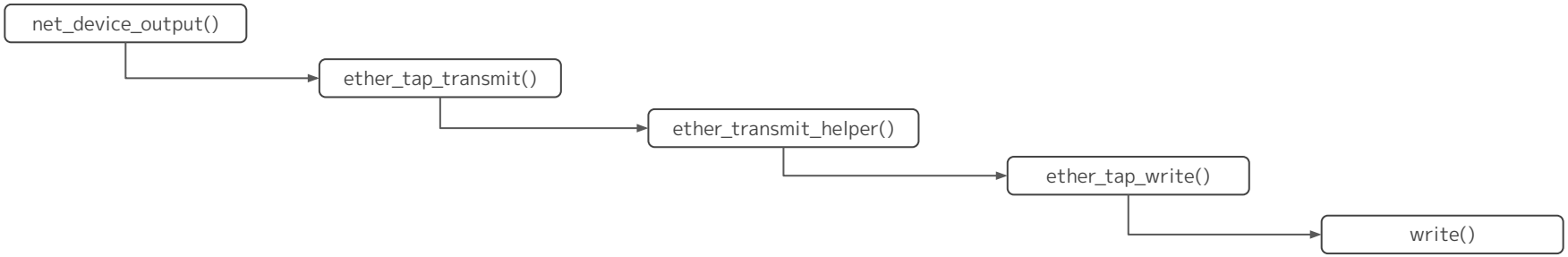
Ethernetデバイス（TAP）送信関数

platform/linux/driver/ether_tap.c

```
static ssize_t
ether_tap_write(struct net_device *dev, const uint8_t *frame, size_t flen)
{
    return write(PRIV(dev)->fd, frame, flen);
}

int
ether_tap_transmit(struct net_device *dev, uint16_t type, const uint8_t *buf, size_t len, const void *dst)
{
    return ether_transmit_helper(dev, type, buf, len, dst, ether_tap_write);
}
```

write() で書き出すだけ

ether_transmit_helper() を呼び出す
・コールバック関数として ether_tap_write() のアドレスを渡す

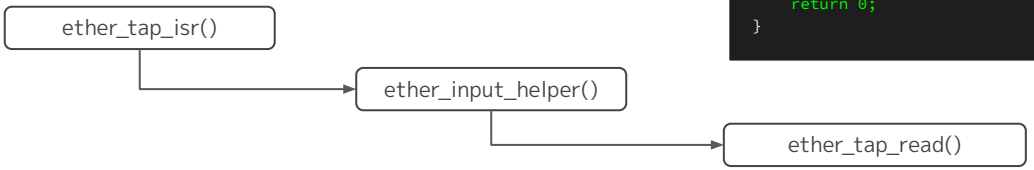


Ethernetデバイス（TAP）割り込みハンドラ

platform/linux/driver/ether_tap.c

```
static ssize_t
ether_tap_read(struct net_device *dev, uint8_t *buf, size_t size)
{
    ssize_t len;

    len = read(PRIV(dev)->fd, buf, size);    read() で読み出すだけ
    if (len <= 0) {
        if (len == -1 && errno != EINTR) {
            errorf("read: %s, dev=%s", strerror(errno), dev->name);
        }
        return -1;
    }
    return len;
}
```

```
static int
ether_tap_isr(unsigned int irq, void *id)
{
    struct net_device *dev;;
    struct pollfd pfd;
    int ret;

    dev = (struct net_device *)id;
    pfd.fd = PRIV(dev)->fd;
    pfd.events = POLLIN;
    while (1) {
        ret = poll(&pfd, 1, 0);    タイムアウト時間を0に設定した poll() で読み込み可能なデータの存在を確認
        if (ret == -1) {
            if (errno == EINTR) {
                continue;    errno が EINTR の場合は再試行（シグナルに割り込まれたという回復可能なエラー）
            }
            errorf("poll: %s, dev=%s", strerror(errno), dev->name);
            return -1;
        }
        if (ret == 0) {    戻り値が 0 だったらタイムアウト（読み込み可能なデータなし）
            /* No frames to input immediately. */
            break;    ループを抜ける
        }
        ether_input_helper(dev, ether_tap_read);    読み込み可能だったら ether_input_helper() を呼び出す
    }                                               ・コールバック関数として ether_tap_read() のアドレスを渡す
    return 0;
}
```



poll(2) - https://man7.org/linux/man-pages/man2/poll.2.html

Ethernetデバイス（TAP）の生成

platform/linux/driver/ether_tap.c

```
struct ether_tap {
    char name[IFNAMSIZ];   TAPデバイスの名前
    int fd;   ファイルディスクリプタ
    unsigned int irq;   IRQ番号
};

...

static struct net_device_ops ether_tap_ops = {
    .open = ether_tap_open,
    .close = ether_tap_close,
    .transmit = ether_tap_transmit,
    .poll = ether_tap_poll,
};

struct net_device *
ether_tap_init(const char *name, const char *addr)
{
    デバイスの生成＆初期化
}
```

```
struct net_device *dev;
struct ether_tap *tap;

dev = net_device_alloc();
if (!dev) {
    errorf("net_device_alloc() failure");
    return NULL;
}                                              デバイスを生成
ether_setup_helper(dev);   Ethernetデバイスの共通パラメータを設定
if (addr) {
    if (ether_addr_pton(addr, dev->addr) == -1) {
        errorf("invalid address, addr=%s", addr);
        return NULL;
    }
}                                              引数でハードウェアアドレスの文字列が渡されたら
                                               それをバイト列に変換して設定する
dev->ops = &ether_tap_ops;   ドライバの関数郡を設定
tap = memory_alloc(sizeof(*tap));
if (!tap) {
    errorf("memory_alloc() failure");
    return NULL;
}
strncpy(tap->name, name, sizeof(tap->name)-1);
tap->fd = -1;   初期値として無効な値（-1）を設定しておく
tap->irq = ETHER_TAP_IRQ;
dev->priv = tap;                               ドライバ内部で使用するプライベートなデータを生成＆保持
if (net_device_register(dev) == -1) {
    errorf("net_device_register() failure");
    memory_free(tap);
    return NULL;
}                                              デバイスをプロトコルスタックに登録
intr_request_irq(tap->irq, ether_tap_isr, INTR_IRQ_SHARED, dev->name, dev);   割り込みハンドラの登録
infof("ethernet device initialized, dev=%s", dev->name);
return dev;
```

テストプログラム

```
> cp test/step11.c test/step12.c
```

step11のテストプログラムをコピーして編集

test/step11.c

```
...
#include "driver/ether_tap.h"
...

static int
setup(void)
{
    ...
    dev = ether_tap_init(ETHER_TAP_NAME, ETHER_TAP_HW_ADDR);
    if (!dev) {
        errorf("ether_tap_init() failure");
        return -1;
    }
    iface = ip_iface_alloc(ETHER_TAP_IP_ADDR, ETHER_TAP_NETMASK);
    if (!iface) {
        errorf("ip_iface_alloc() failure");
        return -1;
    }
    if (ip_iface_register(dev, iface) == -1) {
        errorf("ip_iface_register() failure");
        return -1;
    }
    if (net_run() == -1) {
        errorf("net_run() failure");
        return -1;
    }
    return 0;
}
```

ループバックデバイスはそのままに、新しくEthernetデバイスに関するコードを追記する

Ethernetデバイスの生成

IPインタフェースを生成して紐づける

```
int
main(int argc, char *argv[])
{
    signal(SIGINT, on_signal);
    if (setup() == -1) {
        errorf("setup() failure");
        return -1;
    }
    while (!terminate) {
        sleep(1);
    }
    cleanup();
}
```

プロトコルスタックのセットアップだけして何もせず待機するよう変更

このステップで追加したコードの動作確認

KLab

Makefileを修正してビルド＆実行

Makefile

```
OBJS = util.o \
       ...
       ether.o \

TESTS = test/step0.exe \
        ...
        test/step12.exe \
...

ifeq ($(shell uname),Linux)
...
  DRIVERS := $(DRIVERS) $(BASE)/driver/ether_tap.o
  OBJS := $(OBJS) $(BASE)/intr.o
endif
```

<事前準備> TAPデバイスの生成

```
> sudo ip tuntap add mode tap user $USER name tap0
> sudo ip addr add 192.0.2.1/24 dev tap0
> ip link set tap0 up
```

```
> make
> ./test/step12.exe
22:50:27.285 [I] net_protocol_register: registered, type=0x0800 (net.c:164)
22:50:27.285 [I] ip_protocol_register: registered, type=1 (ip.c:187)
22:50:27.285 [I] net_init: initialized (net.c:268)
22:50:27.285 [I] net_device_register: registered, dev=net0, type=0x0001 (net.c:51)
22:50:27.285 [D] intr_request_irq: irq=36, flags=1, name=net0 (platform/linux/intr.c:33)
22:50:27.285 [D] intr_request_irq: registered: irq=36, name=net0 (platform/linux/intr.c:55)
22:50:27.285 [D] loopback_init: initialized, dev=net0 (driver/loopback.c:116)
22:50:27.285 [I] ip_iface_register: registered: dev=net0, unicast=127.0.0.1, netmask=255.0.0.0, broadcast=127.255.255.255 …
22:50:27.285 [I] net_device_register: registered, dev=net1, type=0x0002 (net.c:51)
22:50:27.285 [D] intr_request_irq: irq=37, flags=1, name=net1 (platform/linux/intr.c:33)
22:50:27.285 [D] intr_request_irq: registered: irq=37, name=net1 (platform/linux/intr.c:55)
22:50:27.285 [I] ether_tap_init: ethernet device initialized, dev=net1 (platform/linux/driver/ether_tap.c:207)
22:50:27.285 [I] ip_iface_register: registered: dev=net1, unicast=192.0.2.2, netmask=255.255.255.0, broadcast=192.0.2.255 …
22:50:27.285 [D] intr_thread: start... (platform/linux/intr.c:71)
22:50:27.285 [D] net_run: open all devices... (net.c:229)
22:50:27.286 [I] net_device_open: dev=net1, state=up (net.c:69)
22:50:27.286 [I] net_device_open: dev=net0, state=up (net.c:69)
22:50:27.286 [D] net_run: running... (net.c:233)
```

2つめのデバイス（net1）として
Ethernetデバイスが登録されている

起動後、開発環境上で別のシェルから ping 192.0.2.2 を実行する

```
22:50:55.364 [D] intr_thread: irq=37, name=net1 (platform/linux/intr.c:89)
22:50:54.331 [D] ether_input_helper: dev=net1, type=0x0806, len=42 (ether.c:114)
        src: 4a:0c:12:da:70:e9
        dst: ff:ff:ff:ff:ff:ff
       type: 0x0806
...
```

Ethernetデバイスがフレームを受信
未登録のプロトコル（0x0806）なので破棄

KLab

STEP 13

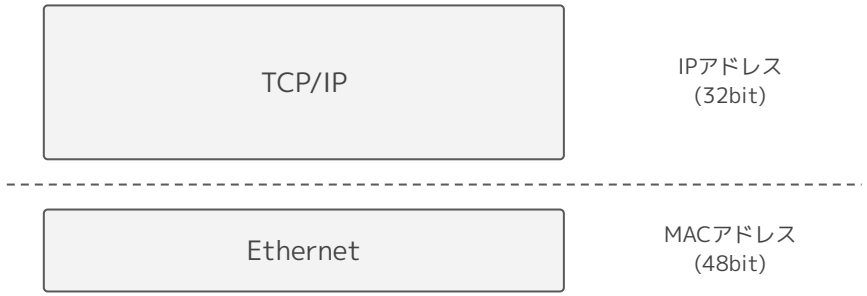
ARP要求への応答

ARP

KLab

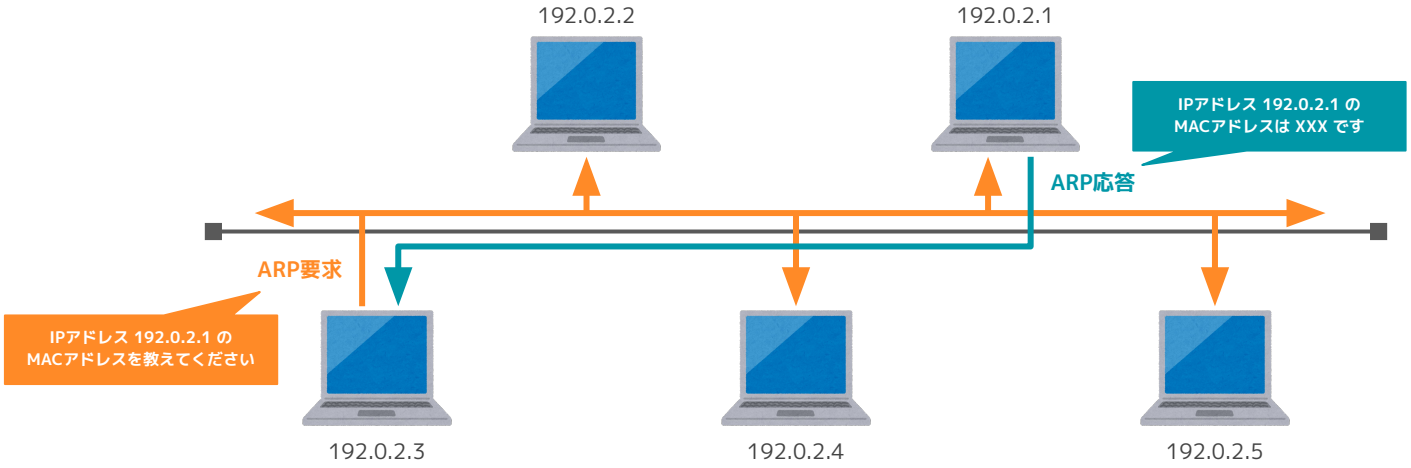
Address Resolution Protocol **RFC 826**

- アドレス解決のためのプロトコル
  - アドレス解決 … ある論理アドレスに紐づいている物理アドレスを得ること
  - ex) 192.0.2.1 というIPアドレスを持つノードのMACアドレスを調べる
- TCP/IPの世界とEthernetなどデータリンクの世界をつなぐ大事なプロトコル



ARPの仕組み

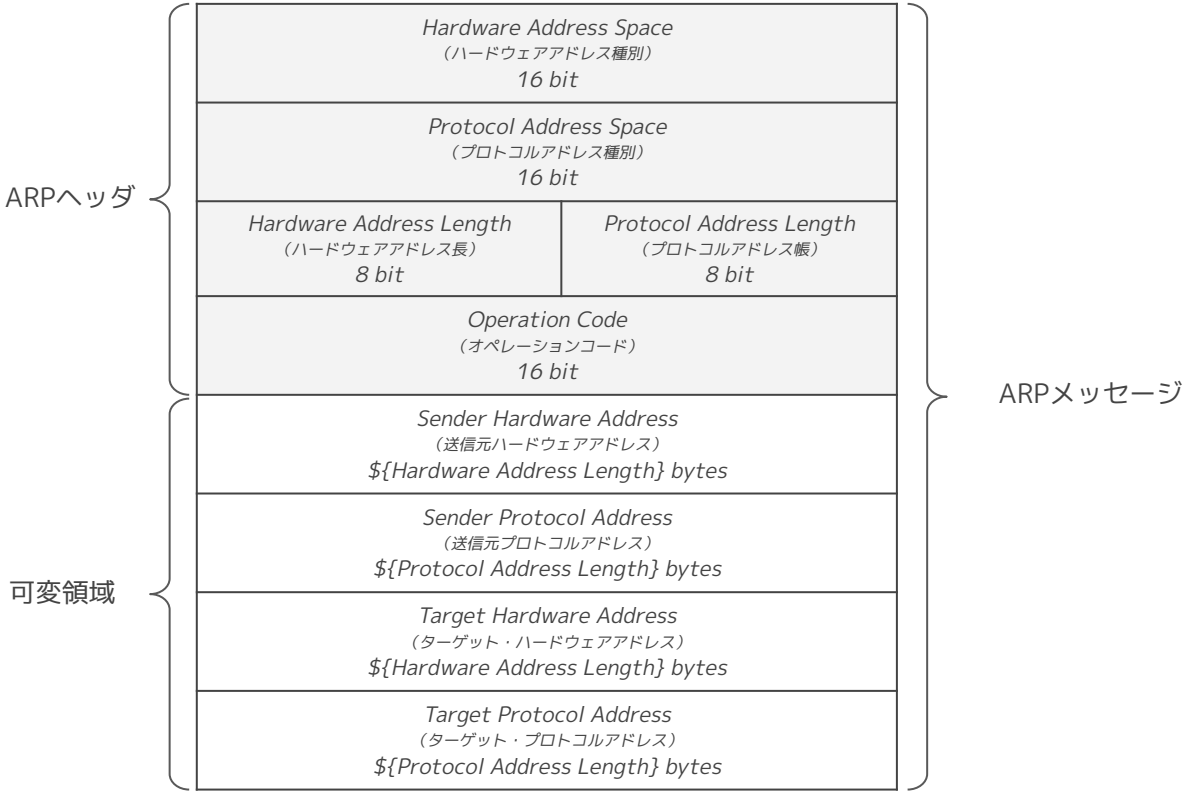
- アドレス解決を必要とするノードがARP要求（ARP Request）をブロードキャストで送信
  - IPアドレスをキーにして紐づくMACアドレスを返答するよう要求
- ARP要求を受け取ったノードのうち対象となるノードがARP応答（ARP Reply）を送信
  - キーとなるIPアドレスに紐づくMACアドレスをユニキャストで回答



前のステップのテストで起きていたこと

1. ping コマンドで 192.0.2.1 へ ICMP メッセージを送信するよう操作
2. Ethernetデバイスから 192.0.2.1 へデータを送信したいが、あて先のMACアドレスが不明
3. ARP要求を送信し、あて先（192.0.2.1）のMACアドレスを問い合わせる
    - FF:FF:FF:FF:FF:FF をあて先としたブロードキャスト送信
4. ARP応答を受信し、あて先のMACアドレスが取得できたら ICMP メッセージを送信
    - ARP応答がないと ICMP メッセージを送信できない（ARP要求を繰り返し送信）
    - このステップでARP応答を返す処理を実装する

ARPメッセージの構造



このステップの作業

KLab

- ARPメッセージの構造体
- デバッグ出力
- ARPの登録とメッセージの受信
- ARP応答の送信

このステップのコードの雛形

```
> git show f0c2a38
```
直近のステップからの差分が表示される

- arp.c **新規**
- arp.h **新規**

ARPメッセージの構造体

arp.c

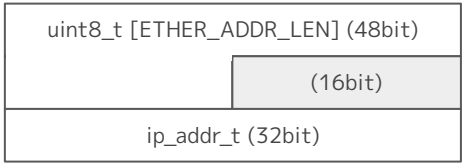
```
/* see https://www.iana.org/assignments/arp-parameters/arp-parameters.txt */
#define ARP_HRD_ETHER 0x0001
/* NOTE: use same value as the Ethernet types */
#define ARP_PRO_IP ETHER_TYPE_IP

#define ARP_OP_REQUEST 1
#define ARP_OP_REPLY   2

struct arp_hdr {
    uint16_t hrd;
    uint16_t pro;
    uint8_t hln;
    uint8_t pln;
    uint16_t op;
};

struct arp_ether_ip {
    struct arp_hdr hdr;
    uint8_t sha[ETHER_ADDR_LEN];
    uint8_t spa[IP_ADDR_LEN];
    uint8_t tha[ETHER_ADDR_LEN];
    uint8_t tpa[IP_ADDR_LEN];
};
```

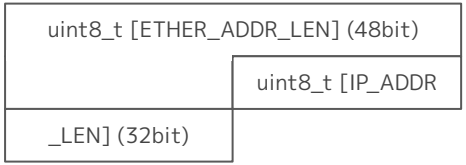
ハードウェアアドレス種別とプロトコルアドレス種別の定数
※ 簡略化のためEthernetとIPだけを対象としている

ARPヘッダの構造体

Ethernet/IP ペアのためのARPメッセージ構造体

spa (tpa) を ip_addr_t にすると sha (tha) とのあいだに
パディングが挿入されてしまうので注意
アラインメント（境界揃え）処理によって 32bit 幅の変数は
4の倍数のアドレスに配置するよう調整されてしまう



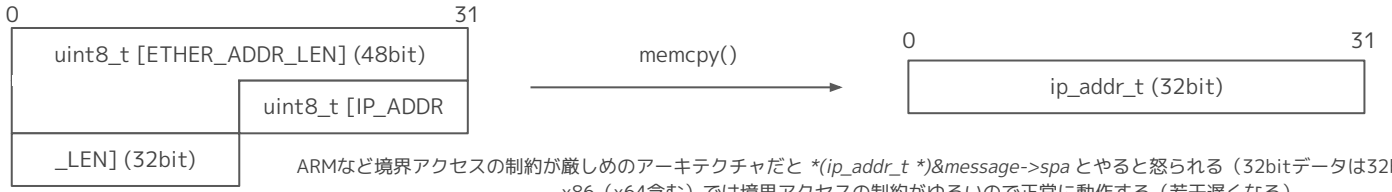


デバッグ出力

arp.c

```
static void
arp_dump(const uint8_t *data, size_t len)
{
    struct arp_ether_ip *message;
    ip_addr_t spa, tpa;
    char addr[128];

    message = (struct arp_ether_ip *)data;  ここでは Ethernet/IP ペアのメッセージとみなす
    flockfile(stderr);
    fprintf(stderr, "        hrd: 0x%04x\n", ntoh16(message->hdr.hrd));
    fprintf(stderr, "        pro: 0x%04x\n", ntoh16(message->hdr.pro));
    fprintf(stderr, "        hln: %u\n", message->hdr.hln);
    fprintf(stderr, "        pln: %u\n", message->hdr.pln);
    fprintf(stderr, "         op: %u (%s)\n", ntoh16(message->hdr.op), arp_opcode_ntoa(message->hdr.op));
    fprintf(stderr, "        sha: %s\n", ether_addr_ntop(message->sha, addr, sizeof(addr)));
    memcpy(&spa, message->spa, sizeof(spa));  spa が uint8_t [4] なので、一旦 memcpy() で ip_addr_t の変数へ取り出す
    fprintf(stderr, "        spa: %s\n", ip_addr_ntop(spa, addr, sizeof(addr)));
    fprintf(stderr, "        tha: %s\n", ether_addr_ntop(message->tha, addr, sizeof(addr)));
    memcpy(&tpa, message->tpa, sizeof(tpa));  tpa も同様に memcpy() で ip_addr_t の変数へ取り出す
    fprintf(stderr, "        tpa: %s\n", ip_addr_ntop(tpa, addr, sizeof(addr)));
#ifdef HEXDUMP
    hexdump(stderr, data, len);
#endif
    funlockfile(stderr);
}
```

ハードウェアアドレス（sha/tha） … Ethernetアドレス（MACアドレス）
プロトコルアドレス（spa/tpa） … IPアドレス



ARMなど境界アクセスの制約が厳しめのアーキテクチャだと *(ip_addr_t *)&message->spa* とやると怒られる（32bitデータは32bit境界でアクセス）
x86（x64含む）では境界アクセスの制約がゆるいので正常に動作する（若干遅くなる）

ARPメッセージの受信

KLab

arp.c

```
static void
arp_input(const uint8_t *data, size_t len, struct net_device *dev)
{
    struct arp_ether_ip *msg;
    ip_addr_t spa, tpa;
    struct net_iface *iface;

    if (len < sizeof(*msg)) {
        errorf("too short");
        return;
    }
    msg = (struct arp_ether_ip *)data;

    debugf("dev=%s, len=%zu", dev->name, len);
    arp_dump(data, len);
    memcpy(&spa, msg->spa, sizeof(spa));
    memcpy(&tpa, msg->tpa, sizeof(tpa));
    iface = net_device_get_iface(dev, NET_IFACE_FAMILY_IP);
    if (iface && ((struct ip_iface *)iface)->unicast == tpa) {
    }
}
```

期待するARPメッセージのサイズより小さかったらエラーを返す

Exercise 13-1: 対応可能なアドレスペアのメッセージのみ受け入れる
(1) ハードウェアアドレスのチェック
　　アドレス種別とアドレス長が Ethernet と合致しなければ中断
(2) プロトコルアドレスのチェック
　　アドレス種別とアドレス長が IP と合致しなければ中断

spa/tpa を memcpy() で ip_addr_t の変数へ取り出す

デバイスに紐づくIPインタフェースを取得する

ARP要求のターゲットプロトコルアドレスと一致するか確認

Exercise 13-2: ARP要求への応答
・メッセージ種別がARP要求だったら arp_reply() を呼び出してARP応答を送信する

ARP応答の送信

arp.c

```
static int
arp_reply(struct net_iface *iface, const uint8_t *tha, ip_addr_t tpa, const uint8_t *dst)
{
    struct arp_ether_ip reply;

    debugf("dev=%s, len=%zu", iface->dev->name, sizeof(reply));
    arp_dump((uint8_t *)&reply, sizeof(reply));
    return net_device_output(iface->dev, ETHER_TYPE_ARP, (uint8_t *)&reply, sizeof(reply), dst);
}
```

Exercise 13-3: ARP応答メッセージの生成
・spa/sha … インタフェースのIPアドレスと紐づくデバイスのMACアドレスを設定する
・tpa/tha … ARP要求を送ってきたノードのIPアドレスとMACアドレスを設定する

デバイスからARPメッセージを送信

ARPの登録

arp.c

```
int
arp_init(void)
{
    Exercise 13-4: プロトコルスタックにARPを登録する
}
```

net.c

```
...
#include "arp.h"

int
net_init(void)
{
    if (intr_init() == -1) {
        errorf("intr_init() failure");
        return -1;
    }
    Exercise 13-5: ARPの初期化関数を呼び出す
    ...
}
```

テストプログラム

```
> cp test/step12.c test/step13.c
```

step12のテストプログラムをコピーする（変更点なし）

このステップで追加したコードの動作確認

Makefileを修正してビルド＆実行

Makefile

```
OBJS = util.o \
       ...
       arp.o \

TESTS = test/step0.exe \
        ...
        test/step13.exe \
```

```
> make
> ./test/step13.exe
```

起動後、開発環境上で別のシェルから ping 192.0.2.2 を実行する

```
...
23:29:39.445 [D] arp_input: dev=net1, len=28 (arp.c:112)
        hrd: 0x0001
        pro: 0x0800
        hln: 6
        pln: 4
         op: 1 (Request)
        sha: 4a:0c:12:da:70:e9
        spa: 192.0.2.1
        tha: 00:00:00:00:00:00
        tpa: 192.0.2.2
```

ARP要求の受信

```
23:29:39.445 [D] arp_reply: dev=net1, len=28 (arp.c:87)
        hrd: 0x0001
        pro: 0x0800
        hln: 6
        pln: 4
         op: 2 (Reply)
        sha: 00:00:5e:00:53:01
        spa: 192.0.2.2
        tha: 4a:0c:12:da:70:e9
        tpa: 192.0.2.1
```

ARP応答の送信

```
...
23:29:39.446 [D] icmp_input: 192.0.2.1 => 192.0.2.2, len=64 (icmp.c:99)
```

ICMP Echo メッセージの受信

```
...
23:29:39.446 [D] icmp_output: 192.0.2.2 => 192.0.2.1, len=64 (icmp.c:129)
```

ICMP Echo Reply メッセージの送信

```
...
23:29:39.447 [E] ip_output_device: arp does not implement (ip.c:262)
23:29:39.447 [E] ip_output: ip_output_core() failure (ip.c:339)
...
```

送信時にARPによるアドレス解決が未実装のため送信エラーが発生している

KLab

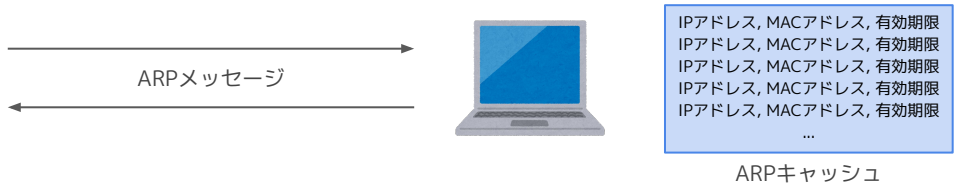
STEP 14

ARPキャッシュ

ARPキャッシュ

ARPによるアドレス解決の結果を一時的に保存（キャッシュ）しておく仕組み

- ARPテーブル（ARP Table）とも呼ばれている
- ARPキャッシュを参照することで解決済みのアドレスを即座に取得できる
- 時間が経過して古くなったキャッシュは自動的に削除される（後のステップで実装）



このステップの作業

- ARPキャッシュの実装
- キャッシュへの反映
- キャッシュの検索

このステップのコードの雛形

```
> git show a36c677
```
直近のステップからの差分が表示される

- arp.c
- arp.h

ARPキャッシュの構造体

arp.c

```
#define ARP_CACHE_SIZE 32

#define ARP_CACHE_STATE_FREE       0
#define ARP_CACHE_STATE_INCOMPLETE 1
#define ARP_CACHE_STATE_RESOLVED   2
#define ARP_CACHE_STATE_STATIC     3

struct arp_cache {
    unsigned char state;
    ip_addr_t pa;
    uint8_t ha[ETHER_ADDR_LEN];
    struct timeval timestamp;
};

static mutex_t mutex = MUTEX_INITIALIZER;
static struct arp_cache caches[ARP_CACHE_SIZE];
```

ARPキャッシュの状態を表す定数

キャッシュの状態

プロトコルアドレス

ハードウェアアドレス

最終更新時刻

ARPキャッシュの構造体

ARPキャッシュの配列（ARPテーブル）

ARPキャッシュの領域確保と削除

KLab

arp.c

```
/*
 * ARP Cache
 *
 * NOTE: ARP Cache functions must be called after mutex locked
 */

static void
arp_cache_delete(struct arp_cache *cache)
{
    char addr1[IP_ADDR_STR_LEN];
    char addr2[ETHER_ADDR_STR_LEN];

    debugf("DELETE: pa=%s, ha=%s", ip_addr_ntop(cache->pa, addr1, sizeof(addr1)), ether_addr_ntop(cache->ha, addr2, sizeof(addr2)));

    /* Exercise 14-1: キャッシュのエントリを削除する
     * ・state は未使用（FREE）の状態にする
     * ・各フィールドを 0 にする
     * ・timestamp は timerclear() でクリアする */
}

static struct arp_cache *
arp_cache_alloc(void)
{
    struct arp_cache *entry, *oldest = NULL;

    for (entry = caches; entry < tailof(caches); entry++) {
        if (entry->state == ARP_CACHE_STATE_FREE) {
            return entry;
        }
        if (!oldest || timercmp(&oldest->timestamp, &entry->timestamp, >)) {
            oldest = entry;
        }
    }
    arp_cache_delete(oldest);
    return oldest;
}
```

ARPキャッシュのテーブルを巡回

使用されていないエントリを返す

空きが無かった時のために一番古いエントリも一緒に探す

現在登録されている内容を削除する

空きが無かったら一番古いエントリを返す

ARPキャッシュの検索と更新

arp.c

```
static struct arp_cache *
arp_cache_select(ip_addr_t pa)
{
    Exercise 14-2: キャッシュの中からプロトコルアドレスが一致するエントリを探して返す
    ・念のため FREE 状態ではないエントリの中から探す
    ・見つからなかったら NULL を返す
}

static struct arp_cache *
arp_cache_update(ip_addr_t pa, const uint8_t *ha)
{
    struct arp_cache *cache;
    char addr1[IP_ADDR_STR_LEN];
    char addr2[ETHER_ADDR_STR_LEN];

    Exercise 14-3: キャッシュに登録されている情報を更新する
    (1) arp_cache_select() でエントリを検索する
      ・見つからなかったらエラー（NULL）を返す
    (2) エントリの情報を更新する
      ・state は 解決済み（RESOLVED）の状態にする
      ・timestamp は gettimeofday() で設定する ※ 使い方がわからなかったら調べること
    debugf("UPDATE: pa=%s, ha=%s", ip_addr_ntop(pa, addr1, sizeof(addr1)), ether_addr_ntop(ha, addr2, sizeof(addr2)));
    return cache;
}
```

ARPキャッシュの登録

arp.c

```
static struct arp_cache *
arp_cache_insert(ip_addr_t pa, const uint8_t *ha)
{
    struct arp_cache *cache;
    char addr1[IP_ADDR_STR_LEN];
    char addr2[ETHER_ADDR_STR_LEN];

    debugf("INSERT: pa=%s, ha=%s", ip_addr_ntop(pa, addr1, sizeof(addr1)), ether_addr_ntop(ha, addr2, sizeof(addr2)));
    return cache;
}
```

Exercise 14-4: キャッシュに新しくエントリを登録する
(1) arp_cache_alloc() でエントリの登録スペースを確保する
　・確保できなかったらエラー（NULL）を返す
(2) エントリの情報を設定する
　・state は 解決済み（RESOLVED）の状態にする
　・timestamp は *gettimeofday()* で設定する ※ 使い方がわからなかったら調べること

gettimeofday(2) - https://man7.org/linux/man-pages/man2/gettimeofday.2.html

ARPキャッシュへの反映

arp.c

```
static void
arp_input(const uint8_t *data, size_t len, struct net_device *dev)
{
    ...
    int marge = 0;  更新の可否を示すフラグ

    ...
    memcpy(&spa, msg->spa, sizeof(spa));
    memcpy(&tpa, msg->tpa, sizeof(tpa));
    mutex_lock(&mutex);  キャッシュへのアクセスをミューテックスで保護
    if (arp_cache_update(spa, msg->sha)) {
        /* updated */
        marge = 1;
    }                                   ARPメッセージを受信したら、まず送信元アドレスのキャッシュ情報を更新する（更新なので未登録の場合には失敗する）
    mutex_unlock(&mutex);  アンロックを忘れずに
    iface = net_device_get_iface(dev, NET_IFACE_FAMILY_IP);
    if (iface && ((struct ip_iface *)iface)->unicast == tpa) {
        if (!marge) {  先の処理で送信元アドレスのキャッシュ情報が更新されていなかったら（まだ未登録だったら）
            mutex_lock(&mutex);
            arp_cache_insert(spa, msg->sha);  送信元アドレスのキャッシュ情報を新規登録する
            mutex_unlock(&mutex);                ミューテックスでの保護を忘れずに
        }
        ...
    }
}
```

アドレス解決を実行する関数

KLab

arp.c

```
int
arp_resolve(struct net_iface *iface, ip_addr_t pa, uint8_t *ha)
{
    struct arp_cache *cache;
    char addr1[IP_ADDR_STR_LEN];
    char addr2[ETHER_ADDR_STR_LEN];

    if (iface->dev->type != NET_DEVICE_TYPE_ETHERNET) {
        debugf("unsupported hardware address type");
        return ARP_RESOLVE_ERROR;
    }
    if (iface->family != NET_IFACE_FAMILY_IP) {
        debugf("unsupported protocol address type");
        return ARP_RESOLVE_ERROR;
    }
    mutex_lock(&mutex);
    cache = arp_cache_select(pa);
    if (!cache) {
        debugf("cache not found, pa=%s", ip_addr_ntop(pa, addr1, sizeof(addr1)));
        mutex_unlock(&mutex);
        return ARP_RESOLVE_ERROR;
    }
    memcpy(ha, cache->ha, ETHER_ADDR_LEN);
    mutex_unlock(&mutex);
    debugf("resolved, pa=%s, ha=%s",
        ip_addr_ntop(pa, addr1, sizeof(addr1)), ether_addr_ntop(ha, addr2, sizeof(addr2)));
    return ARP_RESOLVE_FOUND;
}
```

念のため、物理デバイスと論理インタフェースがそれぞれEthernetとIPであることを確認

ARPキャッシュへのアクセスをmutexで保護（アンロックを忘れずに）

プロトコルアドレスをキーにARPキャッシュを検索

見つからなければ ERROR を返す

見つかったハードウェアアドレスをコピー

見つかったので FOUND を返す

IPデータグラム出力時にアドレス解決

ip.c

```
#include "arp.h"

...

static int
ip_output_device(struct ip_iface *iface, const uint8_t *data, size_t len, ip_addr_t dst)
{
    uint8_t hwaddr[NET_DEVICE_ADDR_LEN] = {};
    int ret;

    if (NET_IFACE(iface)->dev->flags & NET_DEVICE_FLAG_NEED_ARP) {
        if (dst == iface->broadcast || dst == IP_ADDR_BROADCAST) {
            memcpy(hwaddr, NET_IFACE(iface)->dev->broadcast, NET_IFACE(iface)->dev->alen);
        } else {
            errorf("arp does not implement");
            return -1;
        }
    }
    return net_device_output(NET_IFACE(iface)->dev, NET_PROTOCOL_TYPE_IP, data, len, hwaddr);
}
```

Exercise 14-5: arp_resolve() を呼び出してアドレスを解決する
・戻り値が ARP_RESOLVE_FOUND でなかったらその値をこの関数の戻り値として返す

テストプログラム

```
> cp test/step13.c test/step14.c
```

step13のテストプログラムをコピーする（変更点なし）

このステップで追加したコードの動作確認

KLab

Makefileを修正してビルド＆実行

Makefile

```
TESTS = test/step0.exe \
        ...
        test/step14.exe \
```

```
> make
> ./test/step14.exe
...
00:27:45.156 [D] arp_input: dev=net1, len=28 (arp.c:221)
...
00:27:45.156 [D] arp_cache_insert: INSERT: pa=192.0.2.1, ha=4a:0c:12:da:70:e9 (arp.c:177)
00:27:45.156 [D] arp_reply: dev=net1, len=28 (arp.c:195)
...
00:27:45.157 [D] ip_input: dev=net1, iface=192.0.2.2, protocol=1, total=84 (ip.c:242)
...
00:27:45.157 [D] icmp_input: 192.0.2.1 => 192.0.2.2, len=64 (icmp.c:99)
...
00:27:45.157 [D] icmp_output: 192.0.2.2 => 192.0.2.1, len=64 (icmp.c:129)
...
00:27:45.157 [D] ip_output_core: dev=net1, dst=192.0.2.1, protocol=1, len=84 (ip.c:296)
...
00:27:45.157 [D] arp_resolve: resolved, pa=192.0.2.1, ha=4a:0c:12:da:70:e9 (arp.c:268)
00:27:45.157 [D] net_device_output: dev=net1, type=0x0800, len=84 (net.c:134)
...
```

起動後、開発環境上で別のシェルから ping 192.0.2.2 を実行する

ARPキャッシュに登録

アドレス解決に成功

デバイスからICMPメッセージを送信

KLab

STEP 15

ARP要求の送信

ここまでのステップで出来るようになったこと

KLab

- 受信したARP要求に応答する
  - ARP要求の送信元のアドレス情報をARPキャッシュに保存
- ARPキャッシュの情報を用いたアドレス解決
  - ARPキャッシュに保存されていないアドレス情報は解決できない
- 能動的にARP要求を送信する機能が未実装（このステップで実装する）

このステップの作業

- アドレス解決を実行する関数へのコード追加
- ARP要求の送信

このステップのコードの雛形

```
> git show db6b289
```

直近のステップからの差分が表示される

- arp.c

アドレス解決を実行する関数へのコード追加

KLab

arp.c

```
int
arp_resolve(struct net_iface *iface, ip_addr_t pa, uint8_t *ha)
{
    ...
    mutex_lock(&mutex);
    cache = arp_cache_select(pa);
    if (!cache) {
        mutex_unlock(&mutex);
        arp_request(iface, pa);
        return ARP_RESOLVE_INCOMPLETE;
    }
    if (cache->state == ARP_CACHE_STATE_INCOMPLETE) {
        pthread_mutex_unlock(&mutex);
        arp_request(iface, pa); /* just in case packet loss */
        return ARP_RESOLVE_INCOMPLETE;
    }
    memcpy(ha, cache->ha, ETHER_ADDR_LEN);
    mutex_unlock(&mutex);
    ...
}
```

Exercise 15-1: ARPキャッシュに問い合わせ中のエントリを作成

(1) 新しいエントリのスペースを確保
- スペースを確保できなかったら ERROR を返す

(2) エントリの各フィールドに値を設定する
- state … INCOMPLETE
- pa … 引数で受け取ったプロトコルアドレス
- ha … 未設定（なにもしなくてOK）
- timestap … 現在時刻（gettimeofday() で取得）

ARP要求の送信関数を呼び出す

問い合わせ中なので INCOMPLETE を返す

見つかったエントリが INCOMPLETE のままだったらパケロスしているかもしれないので念のため再送する
・タイムスタンプは更新しない

ARP要求の送信関数

arp.c

```
static int
arp_request(struct net_iface *iface, ip_addr_t tpa)
{
    struct arp_ether request;

    Exercise 15-2: ARP要求のメッセージを生成する

    debugf("dev=%s, len=%zu", iface->dev->name, sizeof(request));
    arp_dump((uint8_t *)&request, sizeof(request));

    Exercise 15-3: デバイスの送信関数を呼び出してARP要求のメッセージを送信する
    ・あて先はデバイスに設定されているブロードキャストアドレスとする
    ・デバイスの送信間関数の戻り値をこの関数の戻り値とする
}
```

テストプログラム

```
> cp test/step14.c test/step15.c
```

step14のテストプログラムをコピーして編集する

test/step155.c

```
int
main(int argc, char *argv[])
{
    ip_addr_t src, dst;
    uint16_t id, seq = 0;
    size_t offset = IP_HDR_SIZE_MIN + ICMP_HDR_SIZE;

    signal(SIGINT, on_signal);
    if (setup() == -1) {
        errorf("setup() failure");
        return -1;
    }
    ip_addr_pton("192.0.2.2", &src);
    ip_addr_pton("192.0.2.1", &dst);
    id = getpid() % UINT16_MAX;
    while (!terminate) {
        if (icmp_output(ICMP_TYPE_ECHO, 0, hton32(id << 16 | ++seq), test_data + offset, sizeof(test_data) - offset, src, dst) == -1) {
            errorf("icmp_output() failure");
            break;
        }
        sleep(1);
    }
    cleanup();
}
```

OSから見えているTAPデバイスのIPアドレスを宛先にする

このステップで追加したコードの動作確認

KLab

Makefileを修正してビルド＆実行

Makefile

```
TESTS = test/step0.exe \
        ...
        test/step15.exe \
```

```
> make
> ./test/step15.exe
...
15:54:35.194 [D] icmp_output: 192.0.2.2 => 192.0.2.1, len=28 (icmp.c:133)
       type: 8 (Echo)
       code: 0
        sum: 0x4cfb (0x4cfb)
         id: 59624
        seq: 1
15:54:35.194 [D] ip_output_core: dev=net1, iface=192.0.2.1, protocol=1, len=48 (ip.c:268)
...
15:54:35.194 [D] arp_request: dev=net1, len=28 (arp.c:193)
...
15:54:35.194 [D] arp_resolve: cache not found, pa=192.0.2.1 (arp.c:290)
15:54:35.194 [D] arp_request: dev=net1, len=28 (arp.c:196)
...
15:54:35.195 [D] arp_input: dev=net1, len=28 (arp.c:238)
...
15:54:35.195 [D] arp_cache_update: UPDATE: pa=192.0.2.1, ha=16:df:8a:de:c1:4e (arp.c:142)
15:54:36.194 [D] icmp_output: 192.0.2.2 => 192.0.2.1, len=28 (icmp.c:133)
...
15:54:36.194 [D] ip_output_core: dev=net1, iface=192.0.2.1, protocol=1, len=48 (ip.c:268)
...
15:54:36.194 [D] arp_resolve: resolved, pa=192.0.2.1, ha=16:df:8a:de:c1:4e (arp.c:300)
...
15:54:36.195 [D] ip_input: dev=net1, iface=192.0.2.2, protocol=1, total=48 (ip.c:214)
...
15:54:36.195 [D] icmp_input: 192.0.2.1 => 192.0.2.2, len=28 (icmp.c:99)
       type: 0 (EchoReply)
       code: 0
        sum: 0x54fa (0x54fa)
         id: 59624
        seq: 2
...
```

ARPキャッシュが見つからないので
ARP要求を送信

ARP応答を受信してARPキャッシュ
の内容を更新

アドレス解決に成功

KLab

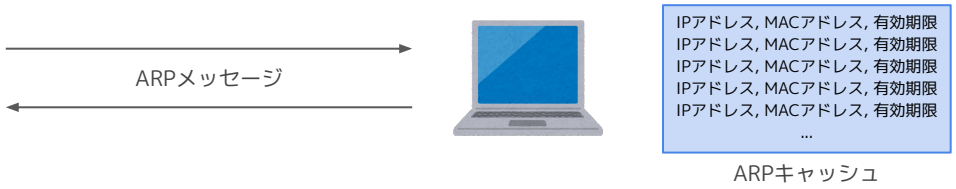
STEP 16

ARPタイマー

ARPキャッシュ

ARPによるアドレス解決の結果を一時的に保存（キャッシュ）しておく仕組み

- ARPテーブル（ARP Table）とも呼ばれている
- ARPキャッシュを参照することで解決済みのアドレスを即座に取得できる
- **時間が経過して古くなったキャッシュは自動的に削除される（後のステップで実装）**



このステップの作業

- タイマー機構の実装
  - タイマーの登録
  - タイマーの確認と発火
  - タイマーのための周期処理
- ARPのタイマーハンドラの登録

このステップのコードの雛形

```
> git show 1b441d3
```
直近のステップからの差分が表示される

- arp.c
- net.c
- net.h
- platform/linux/intr.c

タイマーの登録

net.c

```
struct net_timer {
    struct net_timer *next;
    struct timeval interval;
    struct timeval last;
    void (*handler)(void);
};

...
static struct net_timer *timers;

...

/* NOTE: must not be call after net_run() */
int
net_timer_register(struct timeval interval, void (*handler)(void))
{
    struct net_timer *timer;

    infof("registered: interval={%d, %d}", interval.tv_sec, interval.tv_usec);
    return 0;
}
```

次のタイマーへのポインタ

発火のインターバル

最後の発火時間

発火時に呼び出す関数へのポインタ

タイマーの構造体（リストで管理）

タイマーリスト

Exercise 16-1: タイマーの登録

(1) タイマー構造体のメモリを確保
　・失敗した場合はエラーを返す
(2) タイマーに値を設定
　・最後の発火時間には現在時刻を設定
(3) タイマーリストの先頭に追加

タイマーの確認と発火

net.c

```
int
net_timer_handler(void)
{
    struct net_timer *timer;
    struct timeval now, diff;

    for (timer = timers; timer; timer = timer->next) {
        gettimeofday(&now, NULL);
        timersub(&now, &timer->last, &diff);
        if (timercmp(&timer->interval, &diff, <) != 0) { /* true (!0) or false (0) */

        }
    }
    return 0;
}
```

タイマーリストを巡回

最後の発火からの経過時間を求める

発火時刻を迎えているかどうかの確認

Exercise 16-2: タイマーの発火
・登録されている関数を呼び出す
・最後の発火時間を更新

タイマーのための周期処理

platform/linux/intr.c

```
static int
intr_timer_setup(struct itimerspec *interval)
{
    timer_t id;

    if (timer_create(CLOCK_REALTIME, NULL, &id) == -1) {
        errorf("timer_create: %s", strerror(errno));
        return -1;
    }
    if (timer_settime(id, 0, interval, NULL) == -1) {
        errorf("timer_settime: %s", strerror(errno));
        return -1;
    }
    return 0;
}
```

タイマーの作成

インターバルの設定

```
static void *
intr_thread(void *arg)
{
    const struct timespec ts = {0, 1000000}; /* 1ms */
    struct itimerspec interval = {ts, ts};
...

    debugf("start...");
    pthread_barrier_wait(&barrier);
    if (intr_timer_setup(&interval) == -1) {
        errorf("intr_timer_setup() failure");
        return NULL;
    }
...
        case SIGHUP:
            terminate = 1;
            break;
        case SIGALRM:
            net_timer_handler();
            break;
        default:
...
}

int
intr_init(void)
{
...
    sigaddset(&sigmask, SIGALRM);
    return 0;
}
```

インターバルの値

周期処理用タイマーのセットアップ

周期処理用タイマーが発火した際の処理
・登録されているタイマーを確認するために net_timer_handler() を呼び出す

周期処理用タイマー発火時に送信されるシグナルを追加

ARPのタイマーハンドラの登録

arp.c

```
#define ARP_CACHE_TIMEOUT 30 /* seconds */

static void
arp_timer_handler(void)
{
    struct arp_cache *entry;
    struct timeval now, diff;

    mutex_lock(&mutex);
    gettimeofday(&now, NULL);
    for (entry = caches; entry < tailof(caches); entry++) {
        if (entry->state != ARP_CACHE_STATE_FREE && entry->state != ARP_CACHE_STATE_STATIC) {

        }
    }
    mutex_unlock(&mutex);
}

int
arp_init(void)
{
    struct timeval interval = {1, 0}; /* 1s */

...

    return 0;
}
```

ARPキャッシュへのアクセスをmutexで保護

現在時刻を取得

ARPキャッシュの配列を巡回

未使用のエントリと静的エントリは除外

Exercise 16-3: タイムアウトしたエントリの削除
・エントリのタイムスタンプから現在までの経過時間を求める
・タイムアウト時間（ARP_CACHE_TIMEOUT）が経過していたらエントリを削除する

アンロックを忘れずに

ARPのタイマーハンドラを呼び出す際のインターバル

Exercise 16-4: ARPのタイマーハンドラを登録

テストプログラム

```
> cp test/step14.c test/step16.c
```

step14のテストプログラムをコピーする（変更点なし）

このステップで追加したコードの動作確認

KLab

Makefileを修正してビルド＆実行

Makefile

```
TESTS = test/step0.exe \
        ...
        test/step16.exe \
```

```
> make
> ./test/step16.exe
...
```

起動後、開発環境上で別のシェルから ping -c 1 192.0.2.2 を実行する

```
16:13:42.791 [D] net_run: running... (net.c:304)
...
16:13:48.214 [D] arp_cache_insert: INSERT: pa=192.0.2.1, ha=16:df:8a:de:c1:4e (arp.c:163)
...
```

ARPキャッシュが登録

タイマーが発火するまで（30秒）待つ

```
16:14:19.808 [D] arp_cache_delete: DELETE: pa=192.0.2.1, ha=16:df:8a:de:c1:4e (arp.c:173)
```

ARPキャッシュが削除

KLab

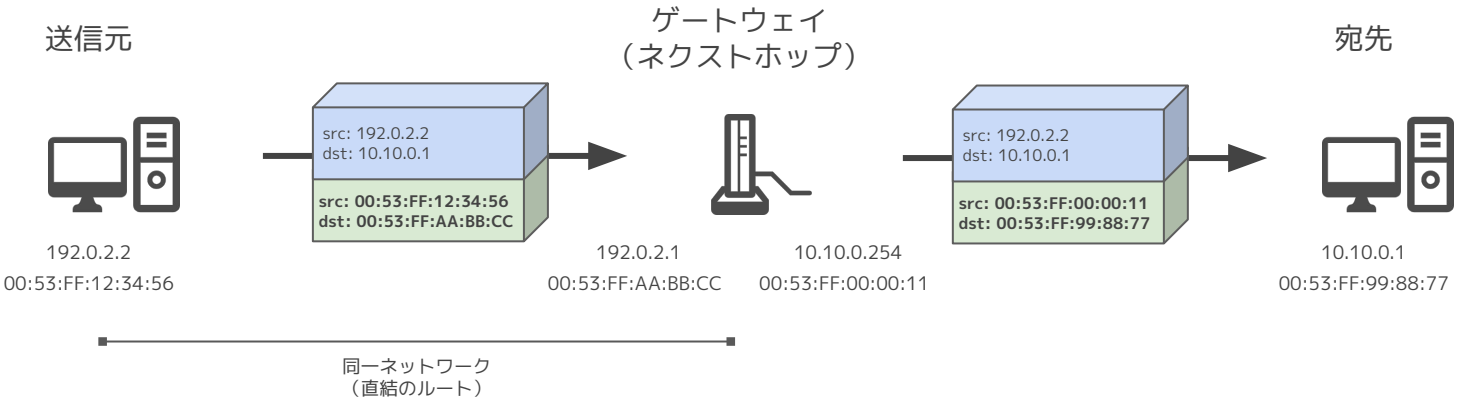
STEP 17

IPルーティングの実装

IPルーティング

ルーティングテーブル

	ネットワークアドレス	サブネットマスク	出力インタフェース	ネクストホップ
直結のルート（自動生成）	127.0.0.0	255.0.0.0	127.0.0.1 (net0)	---
	192.0.2.0	255.255.255.0	192.0.2.2 (net1)	---
デフォルトルート	0.0.0.0	0.0.0.0	192.0.2.2 (net1)	192.0.2.1



このステップの作業

- ルーティングテーブルの実装
- 経路情報の登録
- 経路情報の検索
- デフォルトゲートウェイの登録
- IPデータグラム送信関数への反映

このステップのコードの雛形

```
> git show c124151
```
直近のステップからの差分が表示される

- ip.c
- ip.h

ルーティングテーブルの実装

ip.c

```
struct ip_route {
    struct ip_route *next;      次の経路情報へのポインタ
    ip_addr_t network;          ネットワークアドレス
    ip_addr_t netmask;          サブネットマスク
    ip_addr_t nexthop;          次の中継先のアドレス（なければ IP_ADDR_ANY）
    struct ip_iface *iface;     この経路への送信に使うインタフェース
};                                                          } 経路情報の構造体（リストで管理）

...
static struct ip_route *routes;     経路情報のリスト（ルーティングテーブル）
```

経路情報の登録

KLab

ip.c

```
/* NOTE: must not be call after net_run() */
static struct ip_route *
ip_route_add(ip_addr_t network, ip_addr_t netmask, ip_addr_t nexthop, struct ip_iface *iface)
{
    struct ip_route *route;
    char addr1[IP_ADDR_STR_LEN];
    char addr2[IP_ADDR_STR_LEN];
    char addr3[IP_ADDR_STR_LEN];
    char addr4[IP_ADDR_STR_LEN];

```

Exercise 17-1: 経路情報の登録
・新しい経路情報を作成してルーティングテーブルへ追加する（必要な情報は全て引数で受け取っている）

```
    infof("route added: network=%s, netmask=%s, nexthop=%s, iface=%s dev=%s",
        ip_addr_ntop(route->network, addr1, sizeof(addr1)),
        ip_addr_ntop(route->netmask, addr2, sizeof(addr2)),
        ip_addr_ntop(route->nexthop, addr3, sizeof(addr3)),
        ip_addr_ntop(route->iface->unicast, addr4, sizeof(addr4)),
        NET_IFACE(iface)->dev->name
    );
    return route;
}

...

ip_iface_register(struct net_device *dev, struct ip_iface *iface)
{
...
    if (net_device_add_iface(dev, NET_IFACE(iface)) == -1) {
        errorf("net_device_add_iface() failure");
        return -1;
    }
```

Exercise 17-1: インタフェース登録時にそのネットワーク宛の経路情報を自動で登録する

```
...
}
```

経路情報の検索

ip.c

```
static struct ip_route *
ip_route_lookup(ip_addr_t dst)
{
    struct ip_route *route, *candidate = NULL;

    for (route = routes; route; route = route->next) {
        if ((dst & route->netmask) == route->network) {
            if (!candidate || ntoh32(candidate->netmask) < ntoh32(route->netmask)) {
                candidate = route;
            }
        }
    }
    return candidate;
}

struct ip_iface *
ip_route_get_iface(ip_addr_t dst)
{
    struct ip_route *route;

    route = ip_route_lookup(dst);
    if (!route) {
        return NULL;
    }
    return route->iface;
}
```

ルーティングテーブルを巡回

宛先が経路情報のネットワークに含まれているか確認

この時点で一番有力な候補

サブネットマスクがより長く一致する経路を選択する（ロンゲストマッチ）
・長く一致するほうがより詳細な経路情報となる

ロンゲストマッチで見つけた経路情報を返す

経路情報の中からインタフェースを返す

dst=192.0.2.1

route1 network=192.0.0.0, netmask=255.0.0.0 (/8) … 8bit一致
route2 network=192.0.0.0, netmask=255.255.0.0 (/16) … 16bit一致
route3 network=192.0.2.0, netmask=255.255.255.0 (/24) … 24bit一致

デフォルトゲートウェイの登録

ip.c

```
/* NOTE: must not be call after net_run() */
int
ip_route_set_default_gateway(struct ip_iface *iface, const char *gateway)
{
    ip_addr_t gw;

    if (ip_addr_pton(gateway, &gw) == -1) {
        errorf("ip_addr_pton() failure, addr=%s", gateway);
        return -1;
    }
    if (!ip_route_add(IP_ADDR_ANY, IP_ADDR_ANY, gw, iface)) {
        errorf("ip_route_add() failure");
        return -1;
    }
    return 0;
}
```

デフォルトゲートウェイのIPアドレスを文字列からバイナリ値へ変換

0.0.0.0/0 のサブネットワークへの経路情報として登録する

IPデータグラム送信関数への反映

ip.c

```
ssize_t
ip_output(uint8_t protocol, const uint8_t *data, size_t len, ip_addr_t src, ip_addr_t dst)
{
    struct ip_route *route;
    struct ip_iface *iface;
    char addr[IP_ADDR_STR_LEN];
    ip_addr_t nexthop;
    uint16_t id;

    if (src == IP_ADDR_ANY && dst == IP_ADDR_BROADCAST) {
        errorf("source address is required for broadcast addresses");
        return -1;
    }
    route = ip_route_lookup(dst);
    if (!route) {
        errorf("no route to host, addr=%s", ip_addr_ntop(dst, addr, sizeof(addr)));
        return -1;
    }
    iface = route->iface;
    if (src != IP_ADDR_ANY && src != iface->unicast) {
        errorf("unable to output with specified source address, addr=%s", ip_addr_ntop(src, addr, sizeof(addr)));
        return -1;
    }
    nexthop = (route->nexthop != IP_ADDR_ANY) ? route->nexthop : dst;
    if (NET_IFACE(iface)->dev->mtu < IP_HDR_SIZE_MIN + len) {
        errorf("too long, dev=%s, mtu=%u < %zu",
            NET_IFACE(iface)->dev->name, NET_IFACE(iface)->dev->mtu, IP_HDR_SIZE_MIN + len);
        return -1;
    }
    id = ip_generate_id();
    if (ip_output_core(iface, protocol, data, len, iface->unicast, dst, nexthop, id, 0) == -1) {
        errorf("ip_output_core() failure");
        return -1;
    }
    return len;
}
```

送信元アドレスが指定されない場合，ブロードキャストアドレス宛への送信はできない

宛先アドレスへの経路情報を取得

経路情報が見つからなければ送信できない

インタフェースのIPアドレスと異なるIPアドレスで送信できないように制限（強いエンドシステム）

nexthop … IPパケットの次の送り先（IPヘッダの宛先とは異なる）

nexthop も渡すように変更

IPデータグラム送信関数への反映

ip.c

```
static ssize_t
ip_output_core(struct ip_iface *iface, uint8_t protocol, const uint8_t *data, size_t len, ip_addr_t src, ip_addr_t dst, ip_addr_t nexthop, uint16_t id, uint16_t offset)
{
    ...
    return ip_output_device(iface, buf, total, nexthop);
}
```

dst ではなく nexthop 宛に送信するように

テストプログラム

```
> cp test/step15.c test/step17.c
```

step15 のテストプログラムをコピーする（変更点なし）

test/step16.c

```
static int
setup(void)
{
...
    if (ip_route_set_default_gateway(iface, DEFAULT_GATEWAY) == -1) {
        errorf("ip_route_set_default_gateway() failure");
        return -1;
    }
    if (net_run() == -1) {
        errorf("net_run() failure");
        return -1;
    }
    return 0;
}

...

int
main(int argc, char *argv[])
{
...
    src = IP_ADDR_ANY;
    ip_addr_pton("8.8.8.8", &dst);
...
}
```

デフォルトゲートウェイを登録（192.0.2.1）

送信元アドレスは明示的に指定しない

宛先アドレスをインターネット上のノードのIPアドレスにする

このステップで追加したコードの動作確認

Makefileを修正してビルド＆実行

Makefile

```
TESTS = test/step0.exe \
        ...
        test/step17.exe \
```

<事前準備> IP転送とNATの設定

```
> sudo bash -c "echo 1 > /proc/sys/net/ipv4/ip_forward"
> sudo iptables -A FORWARD -o tap0 -j ACCEPT
> sudo iptables -A FORWARD -i tap0 -j ACCEPT
> sudo iptables -t nat -A POSTROUTING -s 192.0.2.0/24 -o eth0 -j MASQUERADE
```

インターネットへつながっている
ネットワークデバイスを指定

```
> make
> ./test/step16.exe
...                                                    インターネット上のノードへ送信（送信元アドレスは未指定）
04:14:00.261 [D] icmp_output: 0.0.0.0 => 8.8.8.8, len=28 (icmp.c:133)
       type: 8 (Echo)
       code: 0
        sum: 0x45e5 (0x45e5)
         id: 61437
        seq: 2                                         送信元アドレスに 192.0.2.2 が選ばれる
04:14:00.261 [D] ip_output_core: dev=net1, iface=192.0.2.2, protocol=1, len=48 (ip.c:356)
...                                                    nexthop（192.0.2.1）のアドレスを解決
04:14:00.261 [D] arp_resolve: resolved, pa=192.0.2.1, ha=62:4f:fa:0c:a4:2d (arp.c:301)
...
04:14:00.272 [D] ip_input: dev=net1, iface=192.0.2.2, protocol=1, total=48 (ip.c:302)
...                                                    インターネット上のノードからの応答を受信
04:14:00.272 [D] icmp_input: 8.8.8.8 => 192.0.2.2, len=28 (icmp.c:99)
       type: 0 (EchoReply)
       code: 0
        sum: 0x4de5 (0x4de5)
         id: 61437
        seq: 2
...
```

KLab

お疲れさまでした！

ゆっくり休んで明日も頑張りましょう💪