自作 OS をネットワーク対応してみる

Daiki Matsunaga

2018 年 11 月

1 はじめに

この記事は，趣味で作っている簡単な OS[1] にネットワーク機能を載せてみたときの作業記録のようなものです．

2 NIC ドライバの実装

まずはネットワークインターフェースカード (NIC) のドライバを書きます．NIC ドライバが実装すべき関数としては，オープン，クローズと送信 (Tx)，受信 (Rx) があれば最低限なんとかなりそうです．ソースコード 1 に定義を示します．

```
1 struct netdev_ops {
2   int (*open)(int minor);
3   int (*close)(int minor);
4   int (*tx)(int minor, struct pktbuf *pkt);
5   struct pktbuf *(*rx)(int minor);
6 };
```

ソースコード 1　NIC ドライバのインターフェース

今回は Realtek の RTL8139 という Ethernet NIC を用いることにします．RTL8139 を選んだ理由としては，

1. 機能が少なく，仕組みが単純でプログラミングが楽
2. QEMU でサポートされている

が挙げられます．

RTL8139 に関して，以下のような資料があります．

- RTL8139D(L) データシート [2]
- RTL8139(A/B) Programming guide V0.1[3]
- OSDev RTL8139[4]

2.0.1 存在確認

rtl8139_probe()(ソースコード 2) は OS 起動時に一度だけ呼ばれ，RTL8139 の存在を確認します．RTL8139 は PCI バス接続であるため，RTL8139 のベンダ ID とデバイス ID を持つデバイスを検索することで存在確認ができます．

```
1  #define RTL8139_VENDORID 0x10ec
2  #define RTL8139_DEVICEID 0x8139
3
4  DRIVER_INIT int rtl8139_probe() {
5    struct pci_dev *thisdev = pci_search_device(RTL8139_VENDORID,
         RTL8139_DEVICEID);
6    if(thisdev != NULL) {
7      rtl8139_init(thisdev);
8    }
9    return (thisdev != NULL);
10 }
```

ソースコード 2　rtl8139_probe()

2.0.2 初期化

RTL8139 が存在していた場合，rtl8139_init()(ソースコード 3) で初期化を行います．

まず，workqueue を送信用と受信用の 2 つ準備しています．これについては後述します．

その次に，I/O ベースアドレスと割り込みラインの情報を PCI コンフィグレーション空間から読みだしています．どちらもデバイスの制御に必須の情報です．I/O ベースアドレスが分かると，それにオフセットを加算することで RTL8139 の各レジスタへアクセスできるようになります．表 1 に RTL8139 の主なレジスタを示します．完全なレジスタ一覧はデータシートにあります [2]．次の for 文では，IDR0-5 レジスタから MAC アドレスを取得しています，

その後は，PCI バスマスタリング (DMA のような機能) の有効化，パワーオン，ソフトウェアリセット，受信バッファの物理アドレスの設定，割り込みマスクの設定，受信動作の設定，送受信開始と続きます．

最後に，先程ゲットした割り込みライン番号の割り込みハンドラ (rtl8139_inthandler()) を登録し，割り込みを受け付けられる状態にします．

```
1  void rtl8139_init(struct pci_dev *thisdev) {
2    rx_wq = workqueue_new("rtl8139_rx_wq");
3    tx_wq = workqueue_new("rtl8139_tx_wq");
4
5    rtldev.pci = thisdev;
6    rtldev.iobase = pci_config_read32(thisdev, PCI_BAR0);
7    rtldev.iobase &= 0xfffc;
8    rtldev.irq = pci_config_read8(thisdev, PCI_INTLINE);
9    rtldev.rxbuf_index = 0;
10   rtldev.txdesc_head = rtldev.txdesc_tail = 0;
11   rtldev.txdesc_free = TXDESC_NUM;
12   /* 省略 */
13
```

表 1　RTL8139 の主なレジスタ

レジスタ	オフセット	説明
IDR0-5	0x0	ID Register 0-5
TSD0-3	0x10	Transmit Status of Descriptor 0-3
TSAD0-3	0x20	Transmit Start Address of Descriptor 0-3
RBSTART	0x30	Receive (Rx) Buffer Start Address
CR	0x37	Command Register
CAPR	0x38	Current Address of Packet Read
IMR	0x3C	Interrupt Mask Register
ISR	0x3E	Interrupt Status Register
RCR	0x44	Receive (Rx) Configuration Register
CONFIG0-1	0x51	Configuration Register 0-1

```
14    struct ifaddr *eaddr = malloc(sizeof(struct ifaddr)+ETHER_ADDR_LEN);
15    eaddr->len = ETHER_ADDR_LEN;
16    eaddr->family = PF_LINK;
17    for(int i=0; i<ETHER_ADDR_LEN; i++)
18      eaddr->addr[i] = in8(RTLREG(IDR)+i);
19
20    /* 省略 */
21
22    //enable PCI bus mastering
23    u16 pci_cmd = pci_config_read16(thisdev, PCI_COMMAND);
24    pci_cmd |= 0x4;
25
26  #define PCI_CONFIG_ADDR 0xcf8
27  #define PCI_CONFIG_DATA 0xcfc
28    u32 addr = (thisdev->bus<<16) | (thisdev->dev<<11) | (thisdev->func<<8) | (
          PCI_COMMAND) | 0x80000000u;
29    out32(PCI_CONFIG_ADDR, addr);
30    out16(PCI_CONFIG_DATA, pci_cmd);
31
32    //power on
33    out8(RTLREG(CONFIG1), 0x0);
34    //software reset
35    out8(RTLREG(CR), CR_RST);
36    while((in8(RTLREG(CR))&CR_RST) != 0);
37    //set rx buffer address
38    out32(RTLREG(RBSTART), KERN_VMEM_TO_PHYS(rtldev.rxbuf));
39    //set IMR
40    out16(RTLREG(IMR), 0x55);
41    //receive configuration
42    out32(RTLREG(RCR), RCR_ALL_ACCEPT | RCR_WRAP);
```

```
43    //enable rx and tx
44    out8(RTLREG(CR), CR_RE|CR_TE);
45    //setup interrupt
46    idt_register(rtldev.irq+0x20, IDT_INTGATE, rtl8139_inthandler);
47    pic_clearmask(rtldev.irq);
48  }
```

ソースコード 3　rtl8139_init()

2.0.3　パケットバッファ

プロトコルスタックにてパケットを生成する場面では，上位層から下位層へとパケットの先頭にヘッダを付け足していきます．逆に，受信パケットの処理を行う際は，下位層から上位層へとヘッダを取り除いていきます．メモリ領域の先頭アドレスを持ち運ぶのでは，かなりやりずらそうです．

そこで，pktbuf というものを用意します．pktbuf には次のような操作が可能です．

- pktbuf_alloc：指定サイズの pktbuf を生成
- pktbuf_create：指定されたメモリ領域，サイズ，メモリ解放関数で pktbuf を生成
- pktbuf_free：メモリ解放
- pktbuf_get_size：サイズの取得
- pktbuf_reserve_headroom：先頭領域の確保
- pktbuf_add_header：先頭領域の追加
- pktbuf_remove_header：先頭領域の削除
- pktbuf_copyin：メモリ領域をコピー

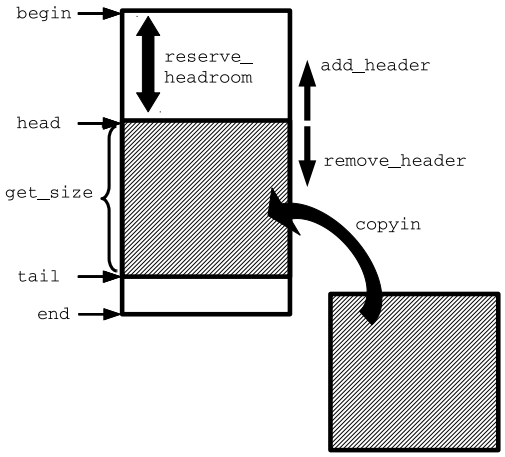

図 1　pktbuf

pktbuf は begin - end，head - tail の 4 つのポインタで管理されており，begin と end はメモリ領域の先頭と末尾を指しています (図 1)．head と tail は，使用中の領域の先頭と末尾を指しており，ヘッダの追加/削除によって変化します．head を加算することでヘッダを削除します．head を追加されうるヘッダサイズだけ加算しておくことで，後々 head を減算しながらヘッダの追加ができます．

なお，メモリ領域の拡張はできないため，あらかじめ十分な量のサイズを確保しておく必要があります．

pktbuf のようなネットワーク用バッファとして，Linux では skb，FreeBSD では mbuf が用意されています [5][7]．共に非常に高機能かつ複雑です．

2.0.4 パケットの送信

パケット送信のインターフェースが，rtl8139_tx() です．送信キューにパケットを入れ，rtl8139_tx_all() の実行を workqueue に依頼します．

workqueue は遅延実行のための機構で，Linux カーネルの同名の機能を参考にしています [6][7]．1 つの workqueue に 1 つカーネルスレッドが対応し，依頼された関数の実行を順次行います．該当スレッドがスケジューリングされるまで関数の実行が遅延されます．

rtl8139_tx_all() は rtl8139_tx_one()(ソースコード 4) を繰り返し呼びます．rtl8139_tx_one() は送信キューからパケットを 1 つ取り出して送信を行います．

RTL8139 には送信ディスクリプタが 4 つあります．各ディスクリプタは 1 つのパケットの送信を管理します．送信の仕方は簡単で，空いている送信ディスクリプタにパケットの先頭物理アドレスとサイズを設定するだけです．割り込みで送信の成功/失敗が通知されます．送信が完了すればそのディスクリプタは次の送信に使えるようになります．4 つのディスクリプタは順繰りに使用していきます．

rtldev.txdesc_free は空き送信ディスクリプタ数を，rtldev.txdesc_head は次の送信ディスクリプタ番号を示します．TSAD0-3 レジスタにはパケットの先頭物理アドレスを，TSD0-3 レジスタの下位 13bit にはパケットサイズを設定します．

```
1  int rtl8139_tx_one() {
2    /* 省略 */
3
4    if(rtldev.txdesc_free > 0) {
5      struct pktbuf *pkt = NULL;
6      if(queue_is_empty(&rtldev.txqueue)) {
7        error = -1;
8        goto out;
9      }
10
11     pkt = list_entry(queue_dequeue(&rtldev.txqueue), struct pktbuf, link);
12
13     out32(RTLREG(TX_TSAD[rtldev.txdesc_head]), KERN_VMEM_TO_PHYS(pkt->head));
14     rtldev.txdesc_pkt[rtldev.txdesc_head] = pkt;
15     out32(RTLREG(TX_TSD[rtldev.txdesc_head]), pktbuf_get_size(pkt));
16     rtldev.txdesc_free--;
17     rtldev.txdesc_head = (rtldev.txdesc_head+1) % TXDESC_NUM;
18     /* 省略 */
19   }
20
21   /* 省略 */
22 }
```

ソースコード 4　パケット送信関連の関数

2.0.5 パケットの受信

パケット受信のインターフェースが，rtl8139_rx() です．受け取ったパケットは最終的に受信キューに入るので，それを待ちます．

パケットが NIC に到着すると，まず割り込みが発生します．割り込みが発生すると，rtl8139_isr()(ソースコード 5) が呼ばれます．割り込みは受信時だけでなく，送信完了時やエラー検出時にも発生します．

rtl8139_isr() では，まず ISR レジスタを読んで割り込みの原因を取得します．次の while 文では，送信ディスクリプタに空きができたかチェックします．

その後は割り込みの原因に応じて処理を行います．送信完了であれば次の送信を依頼，受信完了であれば rtl8139_rx_all()(ソースコード 6) の実行を workqueue に依頼します．

rtl8139_rx_all() では，rtl8139_rx_one() を繰り返し呼び出し，可能であれば一度に複数パケットを受信します．パケットを受信した場合は Ethernet フレーム処理関数 ether_rx() の実行を workqueue に依頼します．ether_rx() は Ethernet フレームを解析して IP などの上位層へとパケットを引き渡します．

workqueue に処理を依頼することで，複数パケットの受信やパケットの処理を割り込みの外で実行できます．それにより，割り込みコンテキストでの実行時間を短縮できます．

rtl8139_rx_one() は 1 つの Ethernet フレームを受信します．受信バッファはリングバッファとして使用されます．rtldev.rxbuf_index で受信バッファの次の読み出し位置を管理しています．受信した Ethernet フレームの情報とサイズは，先頭に付加された 4byte に格納されています．rx_status と rx_size にそれぞれを読みだしています．

その後，エラーチェックを行い，Ethernet フレームをコピーし受信キューへ入れ，次の受信バッファ読み出し位置を更新します．

```
1  void rtl8139_isr() {
2    u16 isr = in16(RTLREG(ISR));
3
4    while(rtldev.txdesc_free < 4 &&
5      (in32(RTLREG(TX_TSD[rtldev.txdesc_tail]))&(TSD_OWN|TSD_TOK)) == (TSD_OWN|
           TSD_TOK)) {
6      struct pktbuf *txed_pkt = rtldev.txdesc_pkt[rtldev.txdesc_tail];
7      if((txed_pkt->flags & PKTBUF_SUPPRESS_FREE_AFTER_TX) == 0) {
8        pktbuf_free(txed_pkt);
9      }
10     rtldev.txdesc_pkt[rtldev.txdesc_tail] = NULL;
11     rtldev.txdesc_tail = (rtldev.txdesc_tail+1) % TXDESC_NUM;
12     rtldev.txdesc_free++;
13   }
14
15   if(isr & ISR_TOK)
16     workqueue_add(tx_wq, rtl8139_tx_all, NULL);
17
18   if(isr & ISR_ROK)
19     workqueue_add(rx_wq, rtl8139_rx_all, NULL);
20
21   if(isr & ISR_TOK)
22     out16(RTLREG(ISR), ISR_TOK);
```

```
23    if(isr & (ISR_FOVW|ISR_RXOVW|ISR_ROK))
24      out16(RTLREG(ISR), ISR_FOVW|ISR_RXOVW|ISR_ROK);
25
26    pic_sendeoi(rtldev.irq);
27  }
```

ソースコード 5　rtl8139_isr()

```
1   void rtl8139_rx_all(void *arg UNUSED) {
2     int rx_count = 0;
3     while(rtl8139_rx_one() == 0)
4       rx_count++;
5
6     if(rx_count > 0) {
7       thread_wakeup(&rtl8139_ops);
8       workqueue_add(ether_wq, ether_rx, (void *)DEVNO(RTL8139_MAJOR,
            RTL8139_MINOR));
9     }
10  }
11
12  int rtl8139_rx_one() {
13    /* 省略 */
14    if((in8(RTLREG(CR)) & CR_BUFE) == 1) {
15      error = -1;
16      goto err;
17    }
18
19    u32 offset = rtldev.rxbuf_index % RXBUF_SIZE;
20    u32 pkt_hdr = *((u32 *)(rtldev.rxbuf+offset));
21    u16 rx_status = pkt_hdr&0xffff;
22    u16 rx_size = pkt_hdr>>16;
23
24    if(rx_status & PKTHDR_RUNT ||
25       rx_status & PKTHDR_LONG ||
26       rx_status & PKTHDR_CRC ||
27       rx_status & PKTHDR_FAE ||
28       (rx_status & PKTHDR_ROK) == 0) {
29      puts("bad packet.");
30      error = -2;
31      goto out;
32    }
33
34    if(!queue_is_full(&rtldev.rxqueue)) {
35      char *buf = malloc(rx_size-4);
36      memcpy(buf, rtldev.rxbuf+offset+4, rx_size-4);
37      struct pktbuf *pkt = pktbuf_create(buf, rx_size-4, free, 0);
38      queue_enqueue(&pkt->link, &rtldev.rxqueue);
39    } else {
40      error = -3;
```

```
41    }
42
43  out:
44    rtldev.rxbuf_index = (offset + rx_size + 4 + 3) & ~3;
45    out16(RTLREG(CAPR), rtldev.rxbuf_index - 16);
46    /* 省略 */
47  }
```

ソースコード 6　パケット受信関連の関数

3　TCP/IP プロトコルスタックの実装

TCP/IP プロトコルスタックは，以前にマイコンで動かすために実装したもの [9] を移植しました (酷い部分が多々あり，かなり手を加えましたが).

受信パケットの処理では，ヘッダを見て適切な処理を行い上位層へ引き渡します．送信パケットは，ヘッダを付けて下位層に委ねます．最終的に Ethernet 処理部が NIC ドライバの送信関数を呼び出すことでパケットが出ていきます.

基本的にはこれだけなので，RFC を読むなりしてひたすら実装していきます．TCP の実装は特別辛かったです (lwIP なり uIP なりを移植するのが良いと思われます).

4　おわりに

TCP/IP プロトコルスタックの解説に関しては力尽きたため，単に RTL8139 ドライバを書いた話になってしまいました.

最終的にはソケットを実装し，システムコールも追加してユーザランドから TCP/UDP で通信できるようになりました．そして，リモートログインをできるようにしてみました.

なお，私の作っている OS”tinyos”のソースコードは GitHub で公開しています.

https://github.com/matsud224/tinyos

参考文献

[1] tinyos https://github.com/matsud224/tinyos

[2] RTL8139(D)L Data Sheet http://www.cs.usfca.edu/~cruse/cs326f04/RTL8139D_DataSheet.pdf

[3] RTL8139(A/B) Programming guide V0.1 http://www.cs.usfca.edu/~cruse/cs326f04/RTL8139_ProgrammersGuide.pdf

[4] OSDev.org https://wiki.osdev.org/Main_Page

[5] Network buffers The BSD, Unix SVR4 and Linux approaches https://people.sissa.it/~inno/pubs/skb-reduced.pdf

[6] オペレーティングシステム II(2010 年) 筑波大学 情報科学類 講義資料 http://www.coins.tsukuba.ac.jp/~yas/coins/os2-2010/

[7] Linux カーネル 2.6 解読室 (ソフトバンククリエイティブ)

[8] Understanding TCP/IP Network Stack & Writing Network Apps https://www.cubrid.org/blog/understanding-tcp-ip-network-stack
[9] tinyip https://github.com/matsud224/tinyip
[10] 詳解 TCP/IP Vol.1 プロトコル (W・リチャード・スティーヴンス, ピアソン・エデュケーション)