LP49 ネットワークプログラムの説明

H_2O

平成 21 年 6 月 15 日

目 次

第1章　LP49のNWプログラムの基本

1.1　はじめに

NW (network) 機能は OS の最重要機能の一つであり、OS 技術者、OS 研究者はそのプログラム構造とロジックを完全にマスターしておくことが望まれる。しかしながら、NW 機能は複雑なだけに、NW 関連プログラム (プロトコルスタックや Erher ドライバなど) は大変に複雑で難解である。学習用 OS として有名な Minix も、NW 関連プログラムは複雑であり、Tanenbaum の Minix 教科書 "Operating System" も NW 関連プログラムの解説は載っていない。Unix や Linux の NW 関連プログラムは、効率を上げるために益々複雑化しており、簡単に学習できるものではない。

しかるに、Plan9 の Plan9 の NW プログラムは 3 回にわたる作り直しを経ているだけに、構造が整理されており、比較的学習もしやすい。そこで、LP49 は Plan9 の NW 関連プログラム構造を引き継ぐこととした。

1.2　Plan9/LP49のNWプログラムの特徴

(1) Plan9 の NW プログラムは 3 回にわたる作り直し (UNIX sytem5 の straem 方式 → マルチスレッド x-kernel 形式 → 現在の方式) を経ており、構造が整理されている。プログラムは、定期的に作り直しをするべきである。

(2) xkernel など優れた研究成果を取り入れており、学習用としても適切である。

(3) Ether ドライバは Ether card 対応に用意する必要があるが、Plan9 用に開発された Ether ドライバを少ない修正で流用できるので、Ether ドライバの開発工数を節約できる。

(4) 各プログラムは、プロトコル対応のモジュール化、共通インタフェース化など。ソフトウェアの部品化の観点からも、よく整理されている。

第2章　IPプロトコルスタックとソフトウェア部品化

2.1　IPサービス

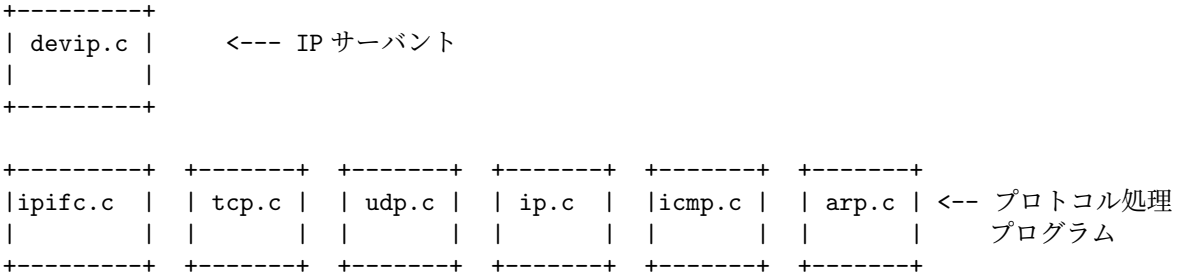
(1) 部品化

NW 処理プログラムは、下図に示すように

- NW サービスのアクセス法を提供する “IP サーバント”
- TCP, UDP, IP 等に対応した、各プロトコル処理プログラム

として、うまく部品化がなされている。

【IP サービスを担当するプログラム】



(2) IP サーバント

- IP サービスは、“IP サーバント” を介して提供される。つまり、LP49-CORE 内で NW 関連のシステムコールは IP サーバントを呼び出すことになる。
- IP サーバントは ”#I”(大文字の’i’) という名前を持ち、名前空間の “/net” にマウントされる。従って、IP サービスは、“/net” 以下の名前空間のファイルを open(), read(), write(),,, することで行われる。例えば、TCP 接続は /net/tcp/clone ファイルを open() することで行われる。
- IP サーバントのソースプログラムは、 “src/9/ip/devip.c” として部品化されている。
- 複雑なプロトコル処理を比較的簡明なプログラムで実現しており、プロトコル処理の教科書としても効果的。

(2) 各層のプロトコル処理プログラム

- プロトコル処理プログラムは、各プロトコル (TCP, UDP, IP, ICMP, ARP 等）対応にプログラム部品化されている。
- NW インタフェース構成 (ローカルアドレス、マスク、MTU サイズ等）を規定するプログラム (ipifc.c) もプロトコル処理プログラムとして統一されている。
- プロトコル処理プログラムは、統一インタフェースを持つ。

- プロトコル処理プログラムのソースは、 src/9/ip/{ipifc.c, tcp.c, udp.c,,,,} である。
- 各プロトコル処理プログラムは有限状態マシンとして実装されており、比較的容易に内容を理解することができる (といっても TCP は機能が複雑なので、プログラムも相応に複雑)。

2.2 Ether サービス

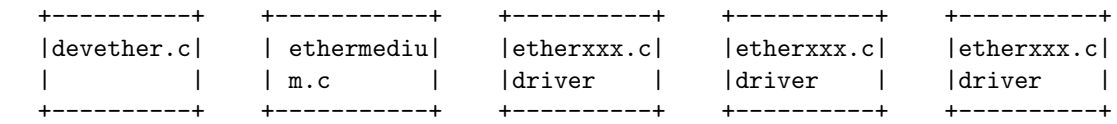
(1) 部品化

Ether ドライバ関連のプログラムは、下図のように

- Ether デバイスのサービスを提供する “Ether サーバント”
- 非同期に到着するパケットを読み出す役目の “ethermedium”
- 各 Ether デバイスに対応したデバイスドライバ

に部品化されている。

【Ether デバイス関連プログラム】



(2) Ether サーバント

- Ether ドライバのサービスは、Ether サーバントを介して行われる。Ether サーバントは、 ”#l”(小文字の’L’) という名前を持ち、名前空間の “/net” にマウントされる。
- Ether サーバントのソースプログラムは、src/9/pc/devether.c である。

(3) Ether ドライバプログラム

各 Ether ドライバプログラムは、src/9/pc/{ether79c970.c, ether8139.c ,,,} として、部品化されている。

第3章　IP サーバント (#I)

3.1　IP サーバントが提供する名前空間

IP サーバントは /net にマウントされて、以下の名前空間と IP サービスを提供する。

```
--- net/ -+-- ipifc/ --+---- clone
          |            |--- stats
          |            |---  0/ -+-- status
          |            :         |-- ctl
          |
          |-- arp
          |-- log
          |-- ndb
          |-- iproute
          |-- ipselftab
          |
          |-- icmp/ ----
          |-- udp/ --+--- clone
          |          |--- status
          |          |---  0/ --+-- data
          |          |          |-- ctl
          |          :          |-- local
          |
          |-- tcp/ --+--- clone
          :          |--- stats
                     |---  0/ --+-- data
                     |          |-- ctl
                     |          |-- local
                     |          |-- remote
                     |          |-- status
                     |          |-- listen
                     |
                     |--- 1/ ---+-- data
                     :          :
```

この名前空間の主要な要素を、以下に説明する。

(1) ipifc

IP インタフェースを管理している。/net/ipifc/clone ファイルをオープンすると、/net/ipifc/N (ここに N は、0, 1, 2,,,) が生成され、/net/ipifc/N/ctl ファイルの記述子が返される。/net/ipifc/N/ctl ファイルには、以下のコマンドを書き込むことによって、インタフェース属性が設定される。

```
bind  ether  <path>
bind  loopbak
add  <localaddress> <mask> <remoteaddress> <mtu>   - - 引数は option
mtu  <n>
```

```
bridge
promiscuous.
....
```

(2) iproute
IP ルーティング情報を管理している。

(3) ipselftab
ローカルアドレスの一覧。

(4) log
ログデータが記録される。

(5) ndb
ネットワーク情報のデータベース。

(6) arp
アドレス解決。

(7) icmp
Internet Control Message Protocol。Ping は、本サービスの一つ。

(8) ip
IP。

(9) ipmux
IP パケットフィルター。

(10) udp
UDP。

(11) tcp
TCP。

3.2 各プロトコルに共通した directory と操作

各プロトコル (TCP, UDP, ICMP, IP 等) は、以下の共通性質を持つ。

1. 各プロトコルのトップ directory は、“clone” ファイル, “stat” ファイル並びに 0, 1, 2,,, と番号名のついた directory を有する。番号名 directory は、そのプロトコルの “論理チャンネル” に対応する。

2. “clone” ファイルを open() すると、“論理チャンネル” が予約され, 番号名 directory が割り当てられ, その下の “ctl ファイル” の記述子が返される。
この論理チャンネルは、リモートマシンに接続要求するとき、あるいリモートマシンからの接続要求を受けるために使われる。

3. 番号名 directory は、“ctl” ファイル、“data” ファイル 他を含む。“ctl” ファイルは制御に、“data” ファイルはデータの送受信に使われる。

4. リモートマシンに接続要求するときは、“ctl” ファイルに connect コマンドを書き込む。
   connect　IP アドレス!ポート

5. リモートマシンからの接続要求受けを宣言するには、“ctl” ファイルに announce コマンドを書き込む。
   announce　ポートまたは＊

6. リモートマシンからの接続要求を受け付けるには、“listen” ファイルを open() して待つ。

3.3 (参考) qsh シェルからのプロトコルスタックの直接制御の例

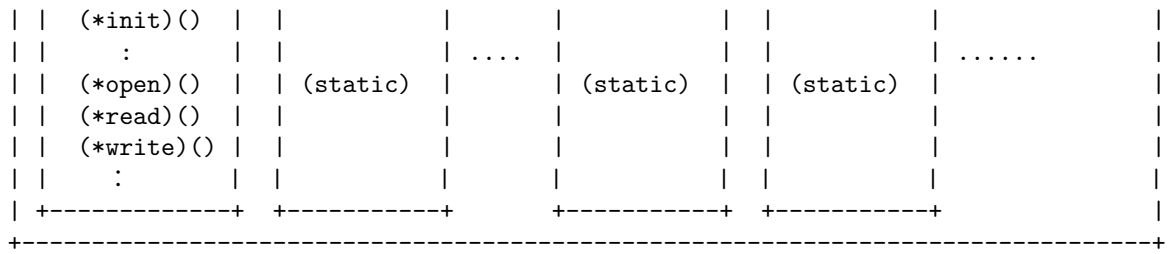
```
LP49[/]: bind -a #l  /net
     →  #l(エル) は ether card である。/net/ether0 が見えるようになる。
LP49[/]: bind -a #I0  /net
     →  #I0(アイゼロ) はプロトコルデバイスである。
        これにより、/net/ipifc,arp.tcp,udp,,,, が見えるようになる。
LP49[/]: open -w /net/ipifc/clone
     →  /net/ipifc/clone を writable (-w) でオープンする。
        インタフェースが生成され (/net/ipifc/0)、その fd が返る。ここでは fd=5 とする。

LP49[/]: write 5 bind  ether  /net/ether0
     →  上記で生成されたファイルに "bind  ether  /net/ether0"命令を書き込む。
        これにより、Ether カードがバインドされる。
LP49[/]: write 5 add 192.168.74.100 255:255:255:0
     →  "add IP アドレス サブネットマスク"命令を書き込む。
        アドレスとサブネットマスクが設定される。
      --- 他のマシンから ping 192.168.74.100 を送ると 応答する ----
LP49[/]: open -w /net/tcp/clone
     →  fd が返る。ここでは 1 とする。
LP49[/]: write 1 announce 80
     --- 他のマシンから telnet 192.168.74.100  80 を受け付けている-----
```

(参考) サーバント dev.c devxxx.c

サーバントのソースプログラムは “devxxx.c” というファイル名をもっている。なお “dev.c” は共通に使われる関数を定義している。

下図に示すように、サーバントのソースプログラム devxxx.c は、xxxopen( ), xxxcreate( ), xxxwrite( ), xxxread( ) 等サーバントのサービスを提供するの関数と、これらの関数へのリンクテーブル (Dev テーブル) を含んでいる。外部からは、 Dev テーブル経由でアクセスされる。サーバントは、適切に設計されたソフトウェアコンポーネントである。

```
+-------------------------------------------------------------------------------+
| +-- Dev ------+  +-----------+       +-----------+  +-----------+              |
| |  type       |  |   init    |       |  open     |  |   read    |              |
| |  name       |  |   func    |       |  func     |  |   runc    |              |
```


(*init)()
:
(*open)()
(*read)()
(*write)()
:
(static)
....
(static)
(static)
.....

第4章　Etherサービス

4.1　Etherサーバント (#l)

Etherデバイスのサービスは、Etherサーバントを介して提供される。Etherサーバントの名前は “#l” (小文字の’L’) であり、名前空間の “/net” にマウントして使われる。ソースプログラムは、src/9/cp/devether.c である。

【Etherサーバントの名前空間】

```
--- net/ -+-- ether0/ ----+---- clone
           |              |--  0/  ----+--- data
           |              :            |--- ctl
           |              :            |--- ifstats
                                       |--- stats
                                       |--- type
```

4.2　Ethermediaプログラム

4.3　Etherドライバプログラム

(1) Etherドライバ #I

第5章　プロトコル処理の流れ

5.1　送信時のデータと処理の流れ

応用プログラムがデータを送信するには, “net/プロトコル/番号/data” ファイルに write() する。net/プロトコル/番号/data ファイルは、IP サーバント (”#I”, devip.c) が提供している。

以下、UDP 送信の場合を 図 5.1 “送信時の処理とデータの流れ” にそって説明する。

なお、以下の説明で”データを引き継ぐ”と書いてある部分は、データのコピーではなく、データブロックのポインタを引き継ぐことを意味する。待ち行列については、後の節で詳しく説明する。

1. NW 接続に対する write() システムコールは、LP49-CORE の中で IP サーバント (”#I” サーバント devip.c) のipwrite() 呼び出しに変換される。

   devip.c の ipwrite() は、所定の処理を行った上で、qwrite(c->wq, a, n) を行って、UDP 層にデータを引き継ぐ。ここで, c->wq は本仮想 UDP 接続の書き出し行列を、qwrite() は待ち行列への書き込み関数である。この様に一見待ち行列を経由して UDP 層にデータを引き継いでいるように見えるが、実はこのc->wq は他の場所でqbypass(udpkick,,) として初期設定されているので、待ち行列による遅延は起こらずに、直ちに UDP 層の udpkick() が呼ばれる。

2. udp.c のudpkick() は、ip.c の ipoput4( ) を呼び出してデータを引き継ぐ。関数名 ipoput4() は IP 層の output で, IP-v4 であることを意味する。

3. ip.c の ipoput4() は、Ether サービスに処理を引き継ぐために、ethermedium.c の etherbwrite() を呼ぶ。

4. etherbwrite() は、Ether サーバント devether.c のetherbwrite( ) を呼ぶ。

5. etherbwrite( ) は、etheroq() を呼ぶ。

6. etheroq( ) は、データを待ち行列に挿入 (qbwrite(ether->oq)) する。ここで待ち行列を使っている理由は、Ether カードがビジーでただちにはパケットを創出できない場合に備えるためである。

7. 次いで 該当 Ether ドライバの transmit( ) を呼ぶ。

8. Ether ドライバの transmit( ) は、txstart() を呼ぶ。

9. txstart( ) は、Ether カードがビジーだったら return する。送るべきパケットは待ち行列に入っている。Ether カードが空きだったら、待ち行列からデータを取り出す。

10. Ether Card にデータを書き込む。これにより、パケットが送信される。

11. Ether Card は送信を完了すると割込みを生じる。txstart() を呼んで, (待ち行列に) 次の送信パケットがあれば処理させる。

このようにUDPの場合の送信は、Etherカードが空きならばLP49-COREの中では待ち合わせることなくパケットが創出される。Etherカードがビジーだった場合には、前のパケットが送り終わったときに割り込みが生じて処理される。

TCPの場合は、フロー制御などが入るのでもっと複雑である。

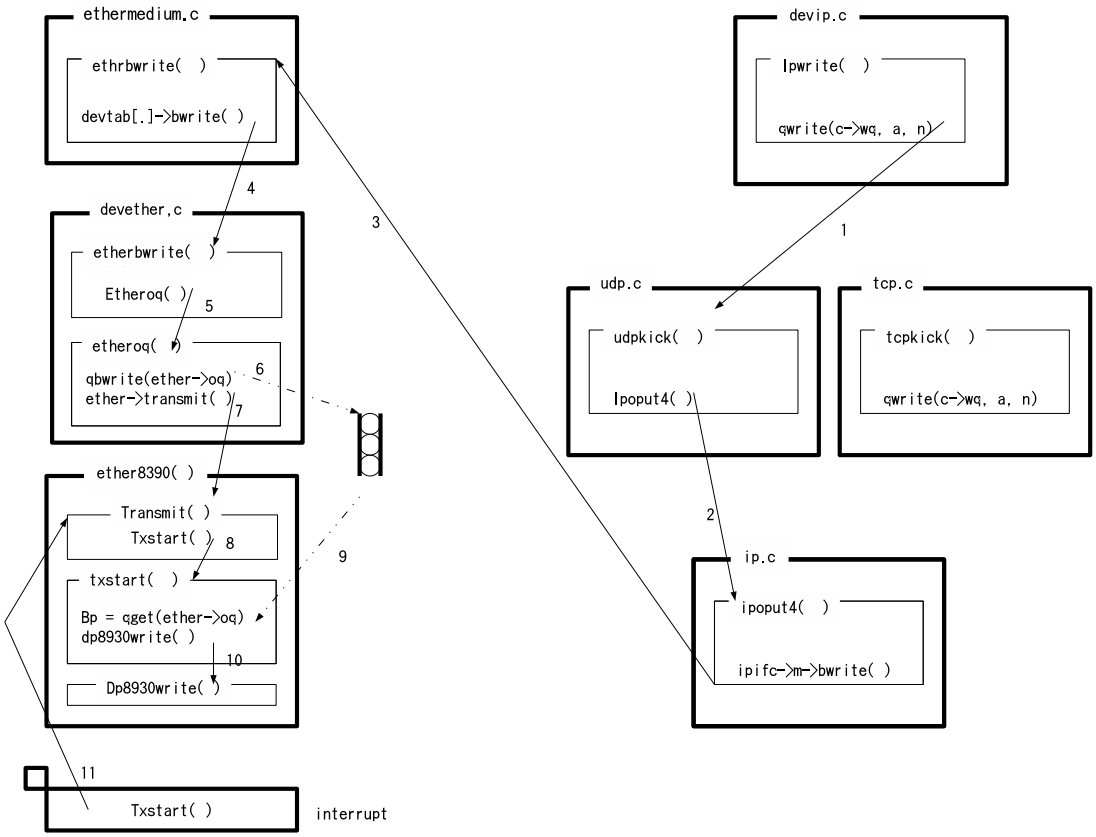


図 5.1: 送信時のデータと処理の流れ

5.2 受信時のデータと処理の流れ

IPサーバント（"#I", devip.c) が提供している "net/プロトコル/番号/data" ファイルをread( )することにより、応用プログラムはデータを受信する。

パケットの到着は、応用プログラムのread()とは非同期に生じるので、この対処が受信処理の興味が沸くところである。本システムでの受信処理の基本は、以下のとおりである。

- "パケット受信スレッド" を用意しておく。具体的には、Etherドライバの接続時に、ethermedium.cの etherbind( ) を呼んで、パケット受信スレッド "etherread4" を生成する。このスレッドは、関数etherread4( ) を 実行する。
- パケットが到着すると、Etherドライバの割り込みハンドラを起動して "パケット待ち行列" に挿入する。

- パケット受信スレッドのetherread4( ) は、パケット待ち行列にパケットが入るのを待つ。このスレッドは、待ち行列にパケットが入ったら、このスレッドを使って IP 層、UDP 層等のプロトコル処理を行う。
- プロトコル処理から応用プロセスへの引き継ぎには、待ち行列を用いる。

以下、UDP 受信の場合を 図 5.2 “受信時の処理とデータの流れ” にそって説明する。

1. パケットが到着すると Ether Card は割込みを発生する。L4 マイクロカーネルは、割り込みをメッセージを割り込みハンドラスレッドに送る。割り込みハンドラスレッドは、該当 Ether ドライバの interrupt() ルーチンを呼ぶ。
2. interrupt( ) は、同一モジュール内のreceive( ) を呼ぶ。
3. receive( ) は、devether.c の etheriq( ) を呼ぶ。
4. etheriq( ) は、受信パケット待ち行列 (netfile->in) に挿入する。ここまでが割込み対処。 (なぜ割込み処理と呼ばなかったか)
5. devether.c の etherbread( ) は、ethermedium.c の パケット受信スレッド (このスレッドは etherread4() を実行している) の下で、待ち行列 (netfile->in) にデータがくるのを待っている。
6. データが到着すると、 ethermedium.c のetherread4( ) (つまり、パケット受信スレッド) にデータが引き継がれる。
7. etherread4( ) は、ip.c の ipiput4( ) を呼び出しデータを引き継ぐ。
8. ipuput4( ) は該当プロトコル (UDP, TCP,,,; ここでは UDP) の udpiput( ) を呼ぶ。
9. udpiput( ) は、qpass( ) を使ってデータを待ち行列 (c->rq) に入れる。
10. devip.c の ipread( ) は、データを待ち行列から取り出したら応用プロセスに返す。

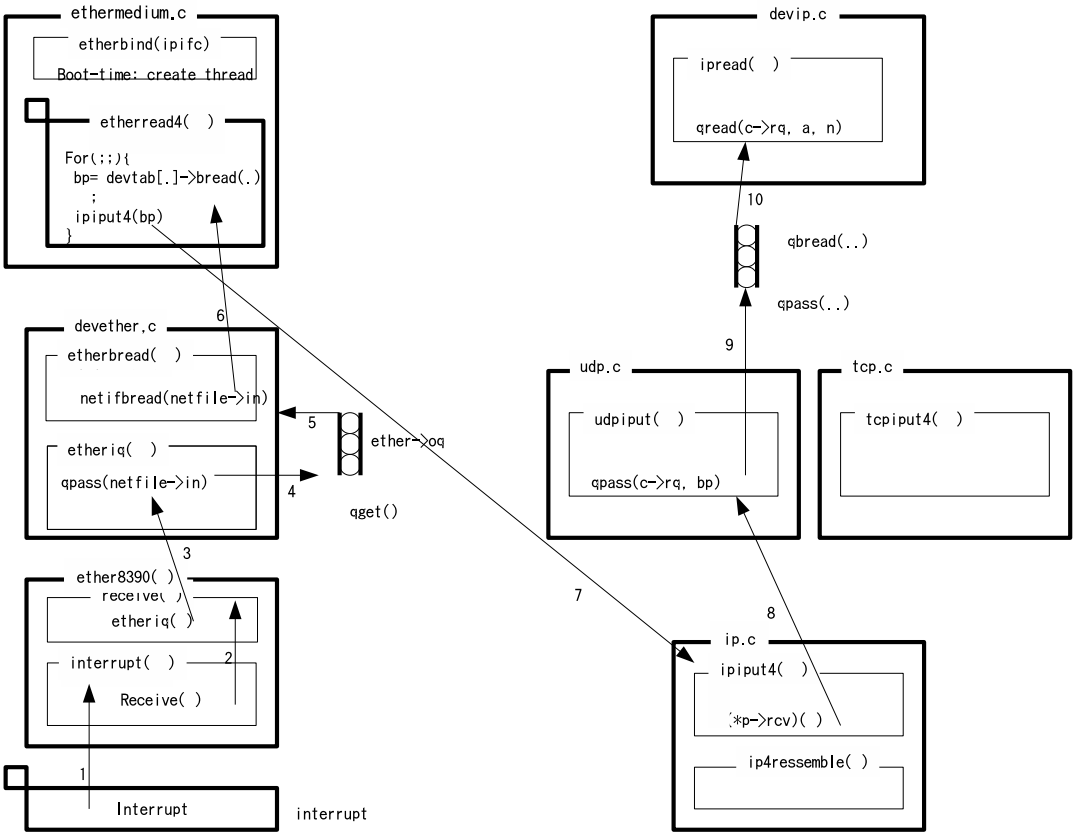


図 5.2: 受信時のデータと処理の流れ

第6章　プロトコルスタックの構造

6.1　基本動作の仕組み

(1) スタック間のコピーレスデータ引き継ぎ

プロトコルスタック間のデータ引き継ぎをコピーレスにすることは、プロトコル処理プログラムの効率化の第一の工夫である。Unix BSD の mbuf 構造体、Linux の skbuf 構造体などがその例である。

Plan9/LP49 では、この目的で Block 構造体と Queue 構造体を使っている。Block 構造体は、データが入っているバッファー域を管理する記述子であり、これを引き継ぐことでコピーレス転送を実現している。

```
struct Block //in portdat.h
{
      Block*  next;     // 待ち行列のリンク
      Block*  list;
      uchar*  rp;       // バッファー域の読み出し点
      uchar*  wp;       // バッファー域の書き込み点
      uchar*  lim;      // バッファー域の終わり＋1
      uchar*  base;     // バッファー域へのポインタ
      void    (*free)(Block*);
      ushort  flag;
      ushort  checksum;
};
```

(2) 待ち行列 (Queue 構造体と qio.c プログラム)

src/9/port/qio.c のプログラムと Queue 構造体 は待ち行列である。qio.c では各種関数が用意されている。Queue 構造体は待ち行列の実体であり、例えば “qbwrite(Queue*, Block*)” により Queue に Block が挿入され、“Block* qbread(Queue*, int)” により、Block が取り出される。

また、待ち行列ではあるが設定により実際には待ち合わせを行わないことも可能である。

Queue 構造体の定義を以下に示す。

```
struct Queue //in qio.c
{
      Lock    _lock;   //%
      Block*  bfirst;  // buffer head
      Block*  blast;   // buffer tail

      int     len;     // bytes allocated to queue
      int     dlen;    // data bytes in queue
      int     limit;   // max bytes in queue
      :
      :
      void    (*kick)(void*); /* restart output */
```

```
        void    (*bypass)(void*, Block*);       /* bypass queue altogether */
        void*   arg;            /* argument to kick */

        QLock   rlock;   // mutex for reading processes
        Rendez  rr;      // process waiting to read
        QLock   wlock;   // mutex for writing processes
        Rendez  wr;      // process waiting to write
        char    err[ERRMAX];
}
```

待ち行列の操作関数（一部）を以下に示す。大部分は、関数名から機能が推定できよう。

```
Block*      qbread(Queue*, int);
long        qbwrite(Queue*, Block*);
Queue*      qbypass(void (*)(void*, Block*), void*); //待ち行列を経ず直接呼び出しさせる
int         qcanread(Queue*);
void        qclose(Queue*);
int         qconsume(Queue*, void*, int);
Block*      qcopy(Queue*, int, ulong);
int         qdiscard(Queue*, int);
void        qflush(Queue*);
void        qfree(Queue*);
int         qfull(Queue*);
Block*      qget(Queue*);
void        qhangup(Queue*, char*);
int         qisclosed(Queue*);
int         qiwrite(Queue*, void*, int);
int         qlen(Queue*);
void        qlock(QLock*);
Queue*      qopen(int, int, void (*)(void*), void*);
int         qpass(Queue*, Block*);
int         qpassnolim(Queue*, Block*);
int         qproduce(Queue*, void*, int);
void        qputback(Queue*, Block*);
long        qread(Queue*, void*, int);
Block*      qremove(Queue*);
void        qreopen(Queue*);
```

次の【Ex.1】は、普通の待ち行列の例を示す。まず、“q1 = qopen(64*1024, Qmsg, 0, 0)” を実行して待ち行列を生成している。送信側のスレッドは、“qbwrite(q1, bk1)” で bk1 を待ち行列 q1 に挿入している。受信側のスレッドは、“bk2 = qbread(q1, size)” を実行して size 分のデータ q1 に挿入されるのを待ち、bk2 を返している。

【Ex.1】

```
Queue *q1;
Block  *bk1, *bk2;
recvque = qopen(64*1024, Qmsg, 0, 0); //Max 64KB までつなげる
 :
n = qbwrite(q1, bk1);  // ブロック bk1 を q1 に挿入してすぐ戻る。
 :
// 別のスレッド
bk2 = qbread(q1, size); // 自動的に待ち合わせをして q1 からブロックを取り出す。
```

次の【Ex.2】は、Queueという名前ではあるが待ち合わせは生じない。まず、“q1 = qbypass(foo, c)”を実行して q2 を生成している。名前 “bypass” から想定されるように、待ち行列はバイパスされ、“q1”にデータが挿入されると、直ちに関数foo( ) が実行されることを意味する。

送信側のスレッドは、“qbwrite(q1, bk1)” で bk1 を待ち行列 q1 に挿入している。これにより、直ちにfoo( ) にブロックが引き継がれて実行される。

【Ex.2】

```
Queue  *q2;
Block  *bk1, *bk2;
void *foo(...)
{ .........
  .........
}
 :
q2 = qbypass(foo, c);
 :
n = qbwrite(q2, bk1);  // bk1 を q2 につないで、foo( ) を呼び出す
 :
```

(3) 送信時の同期のとり方

(4) 受信時の同期のとり方

(5) 送信時の多重化（Multiplexing)

(5) 受信時の分配（Demultiplexing)

・プロトコル種別による分配　（IP 層　→　UDP または　TCP または ....）
・論理チャネルによる場分配　（TCP 層での {rocalIP, localPort, remoteIP, remotePort} による分配。

6.2　主要データの構造

(1) Proto(col) テーブル

```
struct Proto  // Multiplexed プロトコル毎に 1 個
{
      QLock       _qlock;   //%
      char*       name;     //プロトコル名
      int         x;        //プロトコルインデックス
      int         ipproto;  // IP プロトコルタイプ

      char*       (*connect)(Conv*, char**, int);
      char*       (*announce)(Conv*, char**, int);
      char*       (*bind)(Conv*, char**, int);
      int         (*state)(Conv*, char*, int);
      void        (*create)(Conv*);
      void        (*close)(Conv*);
      void        (*rcv)(Proto*, Ipifc*, Block*);
      char*       (*ctl)(Conv*, char**, int);
      void        (*advise)(Proto*, Block*, char*);
      int         (*stats)(Proto*, char*, int);
      int         (*local)(Conv*, char*, int);
      int         (*remote)(Conv*, char*, int);
      int         (*inuse)(Conv*);
      int         (*gc)(Proto*);

      Fs          *f;
      Conv        **conv;    //【注目】conversations の配列
      int         ptclsize;
      int         nc;        //  conversations の総数
      int         ac;
      Qid         qid;       // qid for protocol directory
      ushort      nextport;
      ushort      nextrport;

      void        *priv;
};
```

Proto テーブルは個々のプロトコルを洗わし、TCP, UDP,,, といったプロトコル対応に存在する。(*conncect)(...) から (*gc)(...) は、プロトコル処理関数へのポインタであり、このように全てのプロトコルは統一したインタフェースを持っている。例えばパケットが受信されると、(*rcv)(...) が呼ばれて、所定の処理を行う。

プロトコルの重要役割の一つに多重化がある。つまり、一つの論理回線の中に複数の “論理チャネル” を載せることである。

例えば TCP は多数の TCP 接続を提供する。個々の TCP 接続は、“{ localIPaddress, localPort, remoteIPaddress, remortPort }” により識別される。

個々の “論理チャネル”(接続) を表すのが、次の Conv テーブルである。Proto テーブルは、Conv **conv; フィールドをたどって（一般にはハッシュを使う)、目的の Conv テーブルにアクセスする。

例えば TCP の場合、ハッシュを使ってProto.conv につながっている中から、“{ localIPaddress, localPort, remoteIPaddress, remortPort }” が一致する Conv テーブルを見つける。

(2) Conv(ersation) テーブル

```
struct Conv // 論理接続毎に 1 個
{
      QLock _qlock;
      int     x;      // conversation index
      Proto*  p;      // プロトコルテーブルへ
      ;
      uint    ttl;     // max time to live
      uint    tos;     // type of service
      :
      uchar   laddr[IPaddrlen];  // ローカル IP アドレス
      uchar   raddr[IPaddrlen];  // リモート IP アドレス
      ushort  lport;             // ローカルポート番号
      ushort  rport;             // リモートポート番号
      :
      int     state;
      /* udp specific */
      int     headers;       // data src/dst headers in udp
      :
      Conv*   incall;        // calls waiting to be listened for
      Conv*   next;

      Queue*  rq;    // 受信待ち行列
      Queue*  wq;    // 送信待ち行列（一般に待ち合わせ無）
      Queue*  eq;    // エラーパケット
      Queue*  sq;    // snooping queue
      :
      Rendez  cr;
      char    cerr[ERRMAX];
      :
      Rendez  listenr;
      Ipmulti *multi;       // multicast bindings for this interface
      void*   ptcl;         // protocol specific stuff
      Route   *r;           // last route used
      ulong   rgen;         // routetable generation for *r
};
```

Conv テーブルは “論理チャネル”(接続) を表すテーブルである。(TCP の場合は TCP 接続を意味する。UDP 等の場合は接続というより端点と呼ぶ方がふさわしいかもしれない。)

例えば TCP の場合、ハッシュ技法を使って Proto.conv フィールドから “{ localIPaddress, localPort, remoteIPaddress, remortPort }” が一致する Conv テーブルにアクセスする。

Conv テーブルの laddr, raddr, lport, rport は、それぞれローカル IP アドレス、リモート IP アドレス, ローカルポート番号、リモートポート番号を表す。

Conv テーブルの Queue *rq; Queue *wq; は、それぞれパケットの受信行列、送信行列である。この接続からデータを受けるとるには qbread(rq,..), 送るには qbwite(wq, blkp) を行う。なお、送信の場合は一般には待ち行列はバイパスされ、直ちに指定された関数が実行される。

(3) Fs (ファイルシステム) テーブル

```
struct Fs  // IP プロトコルスタック毎に 1 個
{
      RWlock _rwlock;
      int     dev;
      int     np;
      Proto*  p[Maxproto+1];    /* list of supported protocols */
      Proto*  t2p[256];         /* vector of all protocols */
      Proto*  ipifc;            /* kludge for ipifcremroute & ipifcaddroute */
      Proto*  ipmux;            /* kludge for finding an ip multiplexor */

      IP      *ip;
      Ipselftab       *self;
      Arp     *arp;
      v6params        *v6p;

      Route   *v4root[1<<Lroot];      /* v4 routing forest */
      Route   *v6root[1<<Lroot];      /* v6 routing forest */
      Route   *queue;                 /* used as temp when reinjecting routes */

      Netlog  *alog;

      char    ndb[1024];              /* an ndb entry for this interface */
      int     ndbvers;
      long    ndbmtime;
};
```

6.3 IP サーバント devip (src/9/ip/devip.c)

プロトコルスタックの名前空間を構成し、IP サービスを提供するプログラムである。

(1) 名前空間

Devip サーバントの提供する名前空間 (/net にマウント) を、以下に示す。ここに <N> は論理パス (TCP ならば TCP 接続) を意味し、0，1，2，,,, という数字が使われる。Conv(ersation) テーブルに対応する。

```
--net/ --+--- tcp/ --+--- clone
         |           |--- stats
         |           |--- <N>/ ----+---- ctl     <--- 制御
         |           :             |---- data    <--- データの送受信
         |           :             |---- err
         |                         |---- listen
         |                         |---- local
         |                         |---- remote
         +--- udp/ --+--- clone
         |           |--- stats
         |           |--- <N>/ ----+---- ctl
         |           :             |---- data
         |           :             |---- err
         |                         |---- listen
         |                         |---- local
         |                         |---- remote
         +--- ipifc/ ....
         |
         +--- icmp/  ....
```

```
|
|--- arp
|--- ndb
|--- iproute
|--- ipselftab
|--- log
```

(2) データ構成

Fs，Proto，Conv テーブルの関係を図 6.1 に示す。Fs テーブルは、プロトコルスタックを意味する。Fs の p[プロトコルインデックス] をたぐると、Proto(col) テーブルが求まる。Proto テーブルは、IP, TCP, UDP などといったプロトコルを意味する。Proto テーブルの中身は、connect()，bind()，atate()，rcv() などといったメソッドポインタと、Conv テーブルにアクセスするための配列へのポインタ “Conv **conv” 等である。この配列を論理パス番号 (<N>) でインデックスすると、Conv テーブルが求まる。

Conv テーブルは、論理パス (TCP の場合の TCP 接続など) を表し、local/remote の IP アドレス、local/remote のポート番号等を記録している。データの送受信に関わる重要なフィールドとして、“Queue *rq;” と “Queue *wq;” がある。“rq” はデータを受信する時のアクセス点、“wq” はデータを送信する時のアクセス点を意味する。“Queue” とあるとおり待ち行列の機能を持たせることができる（open 時の設定)。

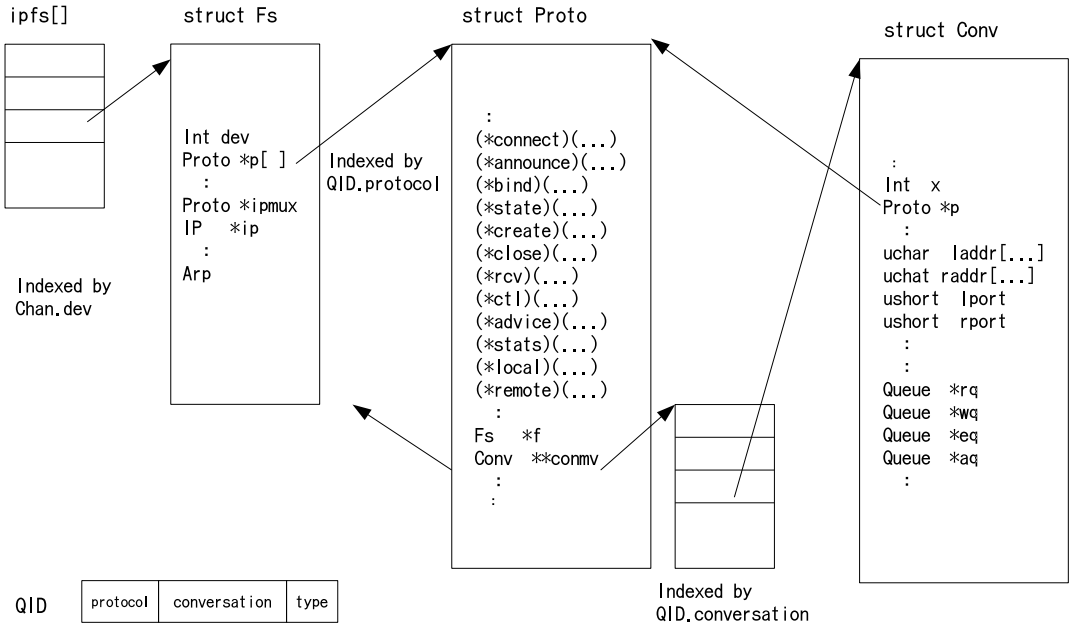


図 6.1: Fs, Proto, Conv テーブルの関係

(3) 主な動作

(1) の名前空間の各要素への操作を実現している。例えば /net/tcp/5/data への書き込みをおこなうと、TCP の該当 Conv テーブルの wq への書き込みが行われる。/net/tcp/5/data への読み出しをおこなうと、TCP の該当 Conv テーブルの rq からの読み出しが行われる。Conv テーブルの rq，wq フィールドは Queue タイプであるが、必ずしも待ち行列経由を意味しない。待ち合わせ同期が不用な場合は、直接引き継がれる。

6.4 IP 層 (src/9/ip/ip.c)

src/9/ip/ip.c は分かりやすいプログラムなので、パケットの送信と受信に関するごく簡単な説明だけを行う。

(1) パケット送信: ipoput4()

例えば UDP 層がパケットを送信する場合には、ip.c の ipoput4() 関数を呼び出す。

ipoput4() は、v4lookup() を呼んで出力方路を求める。電装媒体が Ether の場合は、そして ether-medium.c のetherbwrite() を呼んでパケットを引き渡す。データサイズが制限サイズよりも大きい場合には、フラグメント分けをした上で、各フラグメント毎に繰り返される。

Ether Card ビジーによる待ち合わせは、Ether ドライバ側で対処している。

(2) パケット受信: ipiput4()

Ether カードにパケットが到着すると、割り込みが発生する。Ether ドライバの割り込みハンドラは、到着済みの全パケットを待ち行列 (netif->in) に挿入する。

Ether ドライバ側には、待ち行列 (netif->in) からパケットを読み出して、プロトコル処理を行う専用のスレッドが存在する。“src/9/ip/ethermedium.c” が生成する “etherread4 スレッド” がこれである。このスレッドは, ethermedium.c の etherread4() を実行しており、待ち行列 (netif->in) からパケットを受けとると。“src/9/ip/ip.c” の ipiput4() を呼び出す。

このように入力側のプロトコル処理は、一般に “etherread4 スレッド” の制御下で行われる。

ipiput4() は、受信パケットの宛先IP アドレスが自分か否かを調べる。自分宛でない場合は、v4lookup() を呼んで出力方路を求め, ipoput4() を呼んで転送させる。

自分宛の場合は、もしフラグメント化されていたらまとめ直す。そしてパケットのプロトコル情報を見て、Proto テーブルにアクセスして、(*protocol->rcv)(...) を呼ぶ。(*protocol->rcv)(...) は、UDP の場合は udp.c の udpiput()、TCP の場合は udp.c の tcpiput() である。

6.5 ARP (src/9/ip/arp.c)

6.6 ICMP (src/9/ip/icmp.c)

6.7 UDP 層 (src/9/ip/udp.c)

src/9/ip/ip.c も 分かりやすいプログラムなので、パケットの送信と受信に関するごく簡単な説明だけを行う。

(1) Conv テーブルと送信待ち行列、受信待ち行列

UDP は複数の論理チャネルを多重化している。各論理チャネルは Conv テーブルで表現されている。IP アドレスとポート番号のハッシュを使って、目的の Conv テーブルが求められる。

Conv テーブルが持つ 2 大要素は送信待ち行列 `Queue *wq;` と受信待ち行列 `Queue *rq;` である。この論理チャネルにデータを送るには sq に書き込み、データを受信するには rq から読み出すことになる。

(2) パケット送信: udpkick4()

送信待ち行列 `Queue *wq;` は
"`conv->wq = qbypass(udpkicj, c);`" として初期設定されている。つまり "`wq`" にブロックを挿入する (`qbwrite()`) と、(待ち合わせをせずに直ちに) `udpkick()` 関数が実行される。

`udpkick()` は、所定の検査や処理を行った上で。受けたデータブロックに UDP ヘッダーを追加して、`ip.c` の `ipoput4( )` にパケットを引き継ぐ。これで終わり。

(3) パケット受信: udpiput()

先に説明したように、入力側のプロトコル処理は、"`etherread4` スレッド" の制御下で実行されている。

IP 層が UDP パケットを受信すると、`(*proto->rcv)(...)` を呼び出す。UDP の場合は、`(*proto->rcv)(...)` は `udpiput(...)` である。

`udpiput()` は、IP アドレスとポート番号のハッシュを使って、目的の Conv テーブルが求める。そして Conv テーブルの受信待ち行列にパケットを引き渡す.

```
qpass(conc->rq, bp);
```

6.8 TCP 層 (src/9/ip/tcp.c)

src/9/ip/tcp.c は高度の機能を実現しているので、当然ながら複雑なプログラムである。ここでは、パケットの送信と受信に関するごく簡単な説明だけを行う。

(1) Conv テーブルと送信待ち行列、受信待ち行列

TCP は複数の論理チャネル (TCP 接続) を多重化している。各論理チャネルは Conv テーブルで表現されている。 ハッシュを使って、{localIP, localPort, remoteIP, remotePort} が一致する目的 `Conv` テーブルが求められる。

Conv テーブルが持つ 2 大要素は送信待ち行列 `Queue *wq;` と受信待ち行列 `Queue *rq;` である。この論理チャネルにデータを送るには `wq` に書き込み、データを受信するには `rq` から読み出すことになる。

(2) パケット送信: tcpkick4()

送信待ち行列 `Queue *wq;` は
"`conv->wq = qopen(96*1024, Qkick, tcpkic, c);`" として初期設定されている。

つまり "wq" にブロックを挿入する (`qbwrite()`) と、(待ち合わせをせずに直ちに) `tcpkick()` 関数が実行される。(UDP では、 `qbypass()` を呼んで同様な事を行っていた。)

`tcpkick()` は、Window サイズの処理等を行った上で。`tcpoput4()` に処理を引き継いでいる。

(3) パケット送信: tcpoput44()

`tcpoput4()` は、送信待ち待ち行列からデータブロックを引き継ぎ、TCP ヘッダ等の付加などの処理を行ったうえで、"`ip.c`" の`ipoput4()` を呼んで、IP 層の処理を行わせている。

(4) パケット受信: tcpiput()

先に説明したように、入力側のプロトコル処理は、"`etherread4` スレッド" の制御下で実行されている。

IP 層が TCP パケットを受信すると、`(*proto->rcv)(...)` を呼び出す。TCP の場合は、`(*proto->rcv)(...)` は `tcpiput(...)` である。

`tcpiput()` は、ハッシュを使って、{localIP, localPort, remoteIP, remotePort} が一致する目的 Conv テーブルを求める。そして `Conv` テーブルの受信待ち行列にパケットを引き渡す.

```
qpassnolim(conc->rq, bp);
```

第7章　Etherドライバ

7.1 プログラム構成

7.2 devether サーバント (src/9/pc/devether.c)

7.3 Lance ドライバ (src/9/pc/ether79c970.c)

7.4 8139 ドライバ (src/9/ip/arp.c)

7.5 物理アドレスアクセス