６月１１日のトレースから

RTL-8139 とは？　-- watchdog を探して --

Realtek 社の Ethernet ドライバ定義されているのが、rtl8139 のファイルです。

Ethernet とは何か、簡単に言えば LAN の関係のハードウェアですが、これはかなり情報関係の「常識」に近い用語なので、自分で納得がいくまで調べて下さい。

この rtl-8139 の中でのみ watchdog が定義されています。

一般に、Watchdog は「番犬タイマー」とも呼ばれていますが、
　番犬は、「定期的にエサをもらえると、静かにしている」
　番犬に「エサ」をあげないと、番犬が騒ぎ出す。
　番犬が「エサをもらえない」ということは、飼い主に何か異変が発生した場合である。
　つまり、「番犬が騒ぐ」のは、「飼い主の異変」を検出した場合である。
ということから、外部からの侵入者を監視するというよりも、「飼い主の異変」を察知目的で「番犬タイマー」を使います。

より具体的には、プログラムがフリーズしたり、暴走したりした場合に、番犬タイマーが起動し、これによって、フリーズの原因を取り除いたり、暴走したシステムをリセットしたりして、システムの不安定要因を取り除きます。

以下は、rl_watchdog_f の中身ですが、

```
                if (rep->re_tx_alive)
                {
                        rep->re_tx_alive= FALSE;
                        continue;
                }
                printf("rl_watchdog_f: resetting port %d\n", i);
                printf(
        "TSAD: 0x%04x, TSD: 0x%08x, 0x%08x, 0x%08x, 0x%08x\n",
                        rl_inw(rep->re_base_port, RL_TSAD),
                        rl_inl(rep->re_base_port, RL_TSD0+0*4),
                        rl_inl(rep->re_base_port, RL_TSD0+1*4),
                        rl_inl(rep->re_base_port, RL_TSD0+2*4),
                        rl_inl(rep->re_base_port, RL_TSD0+3*4));
                printf("tx_head %d, tx_tail %d, busy: %d %d %d %d\n",
                        rep->re_tx_head, rep->re_tx_tail,
                        rep->re_tx[0].ret_busy, rep->re_tx[1].ret_busy,
                        rep->re_tx[2].ret_busy, rep->re_tx[3].ret_busy);
                rep->re_need_reset= TRUE;
                rep->re_got_int= TRUE;
                interrupt(rl_tasknr);
        }
```

この Ethernet 関係のみに watchdog が定義されているのは、LAN のポートを通じての送信エラーを検地しているように読めます。
こうしたプログラムを読む場合は、使用されている「変数名」や「メンバー変数名」に着目し、それぞれがどんな意味を持つ変数かを調べていきます。

「送信バッファー」の情報を出力しているのが読み取れるでしょうか？

プロセスの生成部分とプロセスの状態

/usr/src/mm　（メモリ管理）に、forkexit.c というファイルがあります。
このファイルの do_fork()という関数でプロセスの生成に関する処理を行っています。

なぜ､メモリ管理にこの機能があるのでしょうか？｢プロセス｣が生成されたときには、そのプロセスが使用するメモリを新たに割り付ける必要があり、また、｢プロセス管理テーブル｣もメモリにあるために、｢メモリ管理｣で主にこの機能を行い、その結果をカーネルやファイルシステムに通知する構造をとっています。

以下の画面は、do_fork()の関数が終了する前の処理（フォークが成功して戻る前の処理）ですが、９９行目のコメントで､『カーネルとファイルシステムに、(成功した) FORK について通知する｡』と書かれています。メモリ管理で、新たなプロセスにメモリの割付などが成功した後に､その結果をカーネルとファイルシステムに通知しています。

関数 do_fork()が開始するのは３４行目ですが、この９９行目までずっと、メモリの必要量を計算したり、「空き領域」を探したりしています。

```
  rmc->mp_sigstatus = 0;

  /* Find a free pid for the child and put it in the table. */
  do {
        t = 0;                  /* 't' = 0 means pid still free */
        next_pid = (next_pid < 30000 ? next_pid + 1 : INIT_PID + 1);
        for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++)
                if (rmp->mp_pid == next_pid || rmp->mp_procgrp == next_pid) {
                        t = 1;
                        break;
                }
        rmc->mp_pid = next_pid; /* assign pid to child */
  } while (t);

  /* Tell kernel and file system about the (now successful) FORK. */
  sys_fork(who, child_nr, rmc->mp_pid);
  tell_fs(FORK, who, child_nr, rmc->mp_pid);

  /* Report child's memory map to kernel. */
  sys_newmap(child_nr, rmc->mp_seg);

  /* Reply to child to wake it up. */
  setreply(child_nr, 0);
  return(next_pid);             /* child's pid */
```

100 行目に記載されている sys_fork()は、ライブラリに含まれています。/usr/include/minix/syslib の中にあるファイル sys_fork.c で定義されています。

この関数では taskcall をコールし、taskcall では、sendrec でシステム割り込みを発生させていますが、これは、内部コールであるため、「機能番号」には SYSTASK(-2)が割り振られています。

トレースの途中を省略しますが、sys_fork から渡された FORK の結果は、/usr/src/kernel/system.c でポーリングしているプロセス、「sys_task」に手渡されます。

ここで、カーネル内での処理が開始されます。

207 行目、カーネル内では、proc という構造体でプロセスを管理していますが、この構造体に、カーネル内部でのプロセス管理を行っている情報が保管されています。

/usr/src/kernel/proc.h での構造体定義で、どんな情報がカーネルで管理されているか記載されています。

```
_ int p_nr;                     /* number of this process (for fast access) */

  char p_int_blocked;           /* nonzero if int msg blocked by busy task */
  char p_int_held;              /* nonzero if int msg held by busy syscall */
  struct proc *p_nextheld;      /* next in chain of held-up int processes */

  int p_flags;                  /* SENDING, RECEIVING, etc. */
  struct mem_map p_map[NR_SEGS];/* memory map */
  pid_t p_pid;                  /* process id passed in from MM */
  int p_priority;               /* task, server, or user process */

  clock_t user_time;            /* user time in ticks */
  clock_t sys_time;             /* sys time in ticks */
  clock_t child_utime;          /* cumulative user time of children */
  clock_t child_stime;          /* cumulative sys time of children */
```

33 行目の p_nr は、プロセスに直接信号を送るためのプロセス番号、
39 行目の p_flags は、プロセスの状態を表すフラグですが、教科書に記載されている「プロセスの状態」と比較してください。（こうした、プログラム中の記述が、直接教科書の記載と一致する部分は、非常に試験に出しやすいです・・・。）
44 行目では、user time in ticks というコメントの書かれた変数があります。

この、プロセスの状態の状態名は、構造体定義のすぐ下に定義されています。

```
  sigset_t p_pending;           /* bit map for pending signals */
  unsigned p_pendcount;         /* count of pending and unfinished signals */

  char p_name[16];              /* name of the process */
};

/* Guard word for task stacks. */
#define STACK_GUARD     ((reg_t) (sizeof(reg_t) == 2 ? 0xBEEF : 0xDEADBEEF))

/* Bits for p_flags in proc[].  A process is runnable iff p_flags == 0. */
#define NO_MAP          0x01    /* keeps unmapped forked child from running */
#define SENDING         0x02    /* set when process blocked trying to send */
#define RECEIVING       0x04    /* set when process blocked trying to recv */
#define PENDING         0x08    /* set when inform() of signal pending */
#define SIG_PENDING     0x10    /* keeps to-be-signalled proc from running */
#define P_STOP          0x20    /* set when process is being traced */

/* Values for p_priority */
#define PPRI_NONE       0       /* Slot is not in use */
#define PPRI_TASK       1       /* Part of the kernel */
#define PPRI_SERVER     2       /* System process outside the kernel */
#define PPRI_USER       3       /* User process */
#define PPRI_IDLE       4       /* Idle process */
_
```

また、プロセスの実行優先度を表す変数名や定義名も、この部分で探すことができます。

アカウント（時間）情報の記録

/usr/src/kernel/proc.c の 326 行目に、ready()という関数があります。
この関数は、「実行待ち」にあるプロセスを、管理テーブルから拾い出して実行可能に

する処理です。
同様に 369 行目の unready()は、実行状態が終わろうとしているプロセスを、次の実行に備えて「待ち行列」に戻す処理を行っています。

同じファイルにある関数 sched()のコメントを読んでみて下さい。「このプロセスは、長いこと走りすぎたから、他に「実行可能」なプロセスがあればそちらに切り替える」と書かれています。
この sched の名称を一番多数回引用しているのは clock.c ですが、このファイルにある clock_handler()（418 行目）のコメントを見てみると、このプロセスの切り替えに関する設計がわかりやすいと思います。

ここで、proc.h のプロセス構造体のパラメータを見てみると、user time in ticks とか sys time in ticks と書かれている項目があります。
これらは、そのプロセスが MINIX システム上でどれだけの時間稼動していたかを記録しているもので、「時間」に対して「課金」するための構造を持っていることがわかります。

```
  /* If a user process has been running too long, pick another one. */
  if (--sched_ticks == 0) {
        if (bill_ptr == prev_ptr) lock_sched(); /* process has run too long */
        sched_ticks = SCHED_RATE;               /* reset quantum */
        prev_ptr = bill_ptr;                    /* new previous process */
  }
}
```

clock.c の 173 行目に、上のような記述がありますが、ここで、clock.c のタイマ割り込みで、時計の針を刻むように、「カチカチ/tick tack」と時間が刻まれて、「持ち時間」が減っていき、ゼロになると上記のように lock_sched()が呼び出されます。

ここで、プロセスに対して「お前は居座りすぎだ」という警告を出すことになります。

プロセスの管理テーブルは、カーネル内では上記の proc.h の構造体 proc で定義されていますが、メモリ管理（mm）やファイルシステム(fs)ではそれぞれが独立して管理テーブルを持っています。
/usr/src/fs/fproc.h では、ファイルシステム内で使用しているプロセス管理情報を見ることが出来ます。このファイルでは、主にそのプロセスが使用している inode 番号（UNIX のファイルシステムの管理番号）などの情報を扱っています。
/usr/src/mm/mproc.h では、メモリ管理内で使用しているプロセス管理情報を見ることが出来ます。
以下のコメントを読んでみて下さい。

```
/* This table has one slot per process.  It contains all the memory management
 * information for each process.  Among other things, it defines the text, data
 * and stack segments, uids and gids, and various flags.  The kernel and file
 * systems have tables that are also indexed by process, with the contents
 * of corresponding slots referring to the same process in all three.
 */

EXTERN struct mproc {
  struct mem_map mp_seg[NR_SEGS];/* points to text, data, stack */
  char mp_exitstatus;           /* storage for status when process exits */
  char mp_sigstatus;            /* storage for signal # for killed procs */
  pid_t mp_pid;                 /* process id */
  pid_t mp_procgrp;             /* pid of process group (used for signals) */
  pid_t mp_wpid;                /* pid this process is waiting for */
  int mp_parent;                /* index of parent process */
```

プロセス管理に関するソースコードを読む際の参考にして下さい。