カーネル内のトレース（課題トレースの手引き）

カーネル内部に tty_task()という関数があります。

```
/*===========================================================================*
 *                                tty_task                                   *
 *===========================================================================*/
PUBLIC void tty_task()
{
/* Main routine of the terminal task. */

  message tty_mess;             /* buffer for all incoming messages */
  register tty_t *tp;
  unsigned line;

  /* Initialize the terminal lines. */
  for (tp = FIRST_TTY; tp < END_TTY; tp++) tty_init(tp);

  /* Display the Minix startup banner. */
  printf("Minix %s.%s  Copyright 2001 Prentice-Hall, Inc.\n\n",
                                                OS_RELEASE, OS_VERSION);

#if (CHIP == INTEL)
  /* Real mode, or 16/32-bit protected mode? */
#if _WORD_SIZE == 4
  printf("Executing in 32-bit protected mode\n\n");
```

端末処理のメインルーチンであると書かれています。

```
  while (TRUE) {
        /* Check if a timer expired. */
        if (cproc_addr(TTY)->p_exptimers != NULL) tmr_exptimers();

        /* Handle any events on any of the ttys. */
        for (tp = FIRST_TTY; tp < END_TTY; tp++) {
                if (tp->tty_events) handle_events(tp);
        }

        receive(ANY, &tty_mess);

        /* A hardware interrupt is an invitation to check for events. */
        if (tty_mess.m_type == HARD_INT) continue;

        /* Check the minor device number. */
        line = tty_mess.TTY_LINE;
        if ((line - CONS_MINOR) < NR_CONS) {
                tp = tty_addr(line - CONS_MINOR);
        } else
168
```

159 行目に while(TRUE)という記述がありますが、これは「無限ループ」です。一度起動すると、あとはシャットダウンされるまで、ずっと同じ処理を繰り返しています。
FS で send された OS 内部での「割込み」は、この receive でカーネル側に渡されて、処理されます。

要求された処理を実行する、と書かれているのは 204 行目で、

```
        }

        /* Execute the requested function. */
        switch (tty_mess.m_type) {
            case DEV_READ:      do_read(tp, &tty_mess);         break;
            case DEV_WRITE:     do_write(tp, &tty_mess);        break;
            case DEV_IOCTL:     do_ioctl(tp, &tty_mess);        break;
            case DEV_OPEN:      do_open(tp, &tty_mess);         break;
            case DEV_CLOSE:     do_close(tp, &tty_mess);        break;
            case CANCEL:        do_cancel(tp, &tty_mess);       break;
            default:            tty_reply(TASK_REPLY, tty_mess.m_source,
                                          tty_mess.PROC_NR, EINVAL);
        }
    }
}
```

このような画面になっています。
今日の課題では、何を「要求」されていたでしょうか？

ところで、kernel 内部で

```
#grep do_xxxxx *.c
```

を実行して、課題の関数名を探してみると、この tty.c の他にも別のファイルで定義されているのが見つかります。

いずれも、PRIVATE と宣言されていますが、この PRIVATE とは何者でしょうか？

Java のプログラミングでも、scope(適用範囲)という概念があります。Java の場合には、access 修飾子として、public, private, protected などがありました。
Public 宣言されたメソッドは、どこからでもアクセス可能です。そのパッケージが別のプログラムから import されていれば、パッケージの外からでも呼び出されます。
それに対して、private 宣言されたメソッドは、同じクラスの中からしか呼ぶことができません。
Minix のコードでは、C 言語のある予約語（今日の課題の答えになっています）を、Java などの private と同じに見せるために、PRIVATE という定義名をわざわざ定義しています。関数名にこの予約語が使われた場合には、scope は、「そのファイルの中からのみ参照できる」ことになります。（C++は別ですが）C 言語の場合には、クラスの概念がないので、「ファイルの中」ということになります。

ということは、別のファイル内に同じ名前の関数があっても、C のコンパイラには同じファイル内のその名前しか見えていないことになります。　tty.c のファイル内の関数は、同じ tty.c のファイル内でのみ有効である、と考えてよいでしょう。

さて、この関数を調べてみます。

この関数では、handle_events()という関数をコールしています。

実は、この関数の受け渡しで渡されている tty_t という「型」（Java でいうクラスのようなもの）は、「ユーザの応対窓口」を高度に抽象化しています。
コンソール端末で、キーボードや画面を相手にしているだけではなく、通信ポートからリモートログインしてくるユーザがいれば、それらもここで相手をしています。

```
#define TTY_IN_BYTES     256    /* tty input queue size */
#define TAB_SIZE           8    /* distance between tab stops */
#define TAB_MASK           7    /* mask to compute a tab stop position */

#define ESC              '\33'  /* escape */

#define O_NOCTTY       00400    /* from <fcntl.h>, or cc will choke */
#define O_NONBLOCK     04000

typedef _PROTOTYPE( void (*devfun_t), (struct tty *tp) );
typedef _PROTOTYPE( void (*devfunarg_t), (struct tty *tp, int c) );

typedef struct tty {
  int tty_events;               /* set when TTY should inspect this line */

  /* Input queue.  Typed characters are stored here until read by a program. *
  u16_t *tty_inhead;            /* pointer to place where next char goes */
  u16_t *tty_intail;            /* pointer to next char to be given to prog */
  int tty_incount;              /* # chars in the input queue */
  int tty_eotct;                /* number of "line breaks" in input queue */
  devfun_t tty_devread;         /* routine to read from low level buffers */
  devfun_t tty_icancel;         /* cancel any device input */
  int tty_min;                  /* minimum requested #chars in input queue */
  clock_t tty_time;             /* time when the input is available */
```

tty.h というファイルで、この型は定義されています。　devfun_t という「型名」を、「関数のポインタ」として定義しています。
つまり、Java で言えば、どのメソッドを呼び出すか、この「関数のポインタの変数」に代入して実行することができる、ということになります。

```
typedef struct tty {
  int tty_events;               /* set when TTY should inspect this line */

  /* Input queue.  Typed characters are stored here until read by a program. */
  u16_t *tty_inhead;            /* pointer to place where next char goes */
  u16_t *tty_intail;            /* pointer to next char to be given to prog */
  int tty_incount;              /* # chars in the input queue */
  int tty_eotct;                /* number of "line breaks" in input queue */
  devfun_t tty_devread;         /* routine to read from low level buffers */
  devfun_t tty_icancel;         /* cancel any device input */
  int tty_min;                  /* minimum requested #chars in input queue */
  clock_t tty_time;             /* time when the input is available */
  struct tty *tty_timenext;     /* for a list of ttys with active timers */

  /* Output section. */
  devfun_t tty_devwrite;        /* routine to start actual device output */
  devfunarg_t tty_echo;         /* routine to echo characters input */
  devfun_t tty_ocancel;         /* cancel any ongoing device output */
  devfun_t tty_break;           /* let the device send a break */
```

実は、前回のトレースで、カーネルの機能番号に応じて分岐していた部分も、関数のポインタとして定義されていました。

この方法だと、デバイスによって異なる実体の「関数」を代入することができますので、プログラムの構造としては個々のデバイスごとの違いを気にする必要がなくなるのです。
実質的には、ここでは「デバイスドライバ」という、装置を抽象化した関数が呼び出されていると考えることができます。

プリントでは、実際にどの関数が呼ばれたときに write の動作を行うのか、行番号でヒントを記しています。

この変数には、型が一致するならどんな関数でも代入できます。この変数に代入されている「関数の名称」を探せば、実際に「write」という動作が行われている実体部分を見つけることができるでしょう。

プリントでは rs232.c と pty.c の二つを例示しました。
Rs232c は、「シリアルポート」と呼んで、USB がシリアルケーブルの主流となる前に、通信データが入ってくる主な接続ラインでした。現在の LAN のような位置づけでした。
Pty.c は、「擬似ターミナルドライバ」であると書かれています。rlogin（LAN などを経由してのログイン）の場合の処理の流れが、この pty.c のコメントに書かれています。
今日の課題では、「端末に文字列を表示する」機能は、この二つ以外ということになります。

トレース課題の（４）では、handle_events()で、「関数のポインタ変数」に代入されたどの関数がコールされることになるのか、その残り一つの関数を見つけて下さい。
（５）の問題文が最大のヒントです。

プログラムの中では、上で見てきた `while(TRUE)`のような無限ループの構造が、いくつもあります。
FS や MM の main()の中でも、あります。また、kernel の中にも、`yyy_task`()という関数がいくつも待機しています。（OS の仕事の大半は、じっと待つことです。）

そんな中で、プロセスの切り替えなどにも利用される clock_task()がどこにあるか、探してみてください。それが、今日のプリントの最後の「ミニ課題」です。