情報科学部

オペレーティングシステム演習（担当：小林郁夫）　テスト

平成２０年７月１７日

問題１：

以下の文中の（　　）を埋めなさい。　（各１点　１０点）

実行中のプログラムを、（ A：プロセス、タスクも可とした ）と呼ぶ。実行中のプログラムは、様々な資源を利用するが、その代表的なものは（ B：メモリ ）である。画像を読み込めば、画像は内部の（B：メモリ）に展開される。音楽の再生では、音についての情報が（B：メモリ）に展開されている。記録容量が大きいという意味では、ハードディスクのような（ C：周辺、外部記憶も OK とした。 ）機器が有利であるが、（C 周辺[外部記憶]）機器に格納されているデータはそのままでは使用できない。必ずコンピュータ本体の（B：メモリ）に読み込まれて、処理される。コンピュータは、一般に処理の遅い（C：周辺[外部記憶]）機器の動作を待たない。処理の遅い（C：周辺[外部記憶]）機器は、処理が完了すると（ D：割り込み ）を発生し、コンピュータに通知する。コンピュータ本体は、どんな処理をしていても（D：割り込み）を受け付けると、現在実行している（A：プロセス[タスク]）を中断して（D：割り込み）処理プログラムを実行する。この仕組みで、コンピュータは、無駄にキー入力を待って処理を停止するということがなくなる。

（D：割り込み）には、ＯＳ（オペレーティングシステム）の機能を利用するためのソフトウェア（D：割り込み）がある。ソフトウェア（D：割り込み）は、一般に（ E：システム[スーパバイザも OK] ）コールと呼ばれ、コンピュータの資源を有効に利用するための仕組みとして利用される。　[(D)が(C)になっていた。訂正アリ]

また、（D：割り込み）には割り算の分母が（F：０（ゼロ））となった時に発生したり、浮動小数点数の桁があふれた時、すなわち（G：オーバフロー）エラーで発生したりする、いわゆるエラートラップとなるものもある。

こうしたトラップで、システムが安定動作することを保証している。コンピュータに入力されたデータを一時的に蓄えておく領域のことを、バッファと呼ぶ。バッファは、いわゆる（ H：FIFO、悩んだが、QUEUE も OK とした ）型のメモリであって、最初に入力されたデータが最初に取り出される。これとは対照的に、最初に入力されたデータが最後に取り出される（ I：FILO、LIFO も OK ）型のデータ領域がある。関数の戻り先などを記録している（ J：スタック ）などがこれに該当する。

問題２：

以下の用語を解説しなさい。　（各３点　１５点）

解答欄が２行しかなかった点に注意。ネットを参照して要点を抜き出す際に、キーワードをピックアップせず、無関係な記述があった場合には、点はあまり出ていない。

中心となるキーワードに対して２点、以下、付帯的なキーワードで、理解していると思える記述に加点した。

（１）　ウォッチドッグタイマ

システムの暴走[不正な動作、ハングアップ、フリーズなども OK]を監視し検出する。（２点）タイマ（１点）のカウントダウンがゼロになると、リセットなど回復措置を行う。

（２）　フォーク

親プロセスが自分をコピー（１点）して、プロセスを分岐、生成する（２点）。生成されたプロセスは独立して（１点）動作する。「独立」という語がなくても、「親」と「子」と書かれていたら「独立」に見なして採点した。

（３）　パイプ

プロセス間通信の機能（２点）。シェルで複数のプロセスの標準入出力を結合（１点）したり、API で複数プロセス間にデータの受け渡し（１点）の口を作ったりする。

（４）　マッピング

仮想資源に物理資源を割り当てること（３点）。「仮想資源、物理資源」という記述ではなく、「特定の資源、特定の情報」などという記述も、「仮想資源、物理資源」を指すものとして採点した。マッピングを行うものには、メモリ、キーボード、ファイル、プロセス、IP アドレスなど、多種がある。「仮想資源」の語がなくとも、これらいずれかが記述してあれば OK とした。逆に、漠然と「割り当てること」とだけが書かれていた場合には、１点しか出していない。

OS の授業なので、３D グラフィクスのテクスチャー・マッピングの説明だけの場合には、本当は点を出したくなかったが、１点だけ出した。

（５）　アカウント

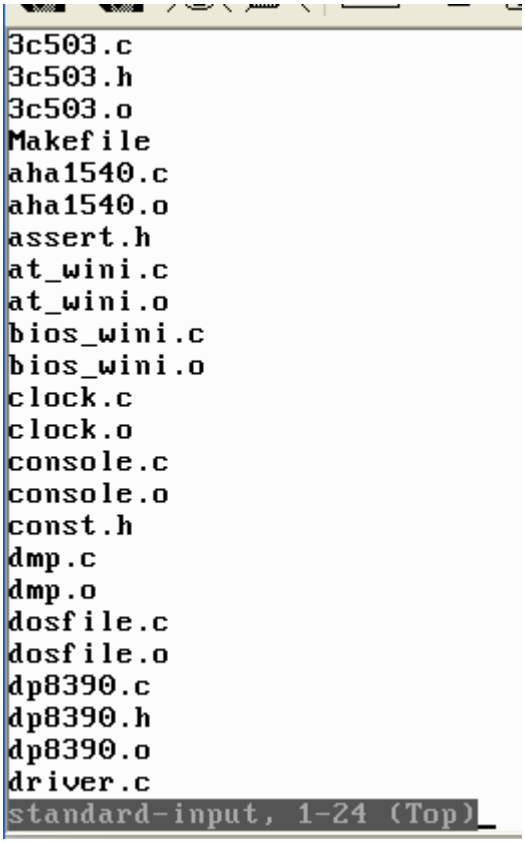
システムの資源を利用するための権利のこと、またはそのための ID を指す。（３点）「システムの資源」を、「コンピュータやネットワーク」などと記載したものも OK とした。

問題３：

以下の設問に答えなさい。実際に試して良い。

（各４点　８点）

（１）　ディレクトリ内にあるファイル名の一覧を、アルファベット順に並べ替えて、１画面ずつ切り替えて表示したい。（右図上）のような表示を得るためのコマンドを、１行で入力したい。コマンドを答えなさい。




```
ls | sort | more      または
ls -1 | more      または
ls -a | more      または
ls | more  （４点）
```

ls 自体が通常はアルファベット順に並べてくれるため、実は、sort を指定しなくても結果は同じになる。

ls | sort

ls -1

の二つは、１ページずつの区切りにはなっていない。（２点）

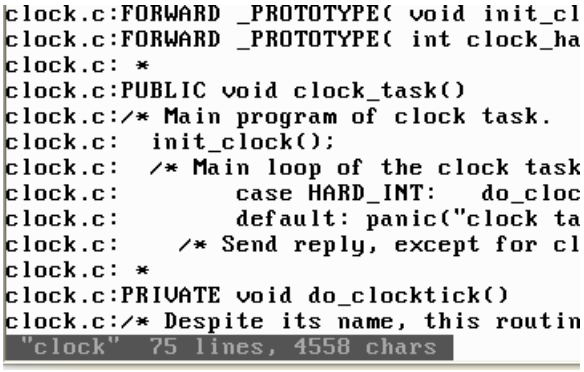
その他は、ls があれば、１点としたが、ls がないものには、点は出さない。

「結果オーライ」だが、パイプの最終段に ls があるものは、ls 自身が標準入力を無視するため、それ以前を無視した結果のコマンドとして採点した。

（2） プログラム中に使われている clock という文字列を全て検索したいが、あまりにも数が多いため、一旦 clock という名前のファイルに落とし、右図下のように、ファイルを開いてじっくり調べたい。ファイルは

vi clock

で開くものとして、ファイルを作成する時に、入力するコマンドを答えなさい。



grep clock *.c > clock　　（４点）
リダイレクションの理解を試す問題だったが、

grep clock *.c
だけでも３点とした。

grep も vi も、ls も、演習を通じて（レポートを作成する際にも）何度も使ったコマンドのはずなので、これ以外にはオマケはできないので、悪しからず。

grep clock *.c | cat > clock　　は、OK。標準入出力をよく理解している。
Grep clock *.c | vi clock　は、惜しい。結果オーライだけれど、「偶然」標準入力に i という文字が入らない限りは動作しないため、満点は出せない。（あまりいい解答ではない。）
grep clock *.c | vi clock.c
も惜しいが、clock.c のファイルを壊してしまうため、オマケして２点。

問題４：

ファイルシステム(/usr/src/fs)で、do_write という関数をコールすると、関数 read_write がコールされ、read_write から rw_chunk がコールされ、rw_chunk から ra_head がコールされる。

この関係を、do_write → read_write → rw_chunk → ra_head と記したとする。

関数 do_open をコールすると、dev_io, get_block, get_block2, rw_inode, common_open などがコールされるが、これらは、それぞれどれがどれをコールする関係になっているか。上記の例にならって、呼び出し関係を記しなさい。（完全解答１０点）

do_open → common_open → rw_inode → get_block → get_block2 → dev_io　（10 点）
関数が一箇所だけ誤挿入されているものは、４点減点で６点とした。例：
do_open → common_open → rw_inode → dev_io → get_block → get_block2　（６点）
また、途中まで書かれているものは、
do_open → common_open → rw_inode → get_block →
で２点とした。（完全解答をベースに、関数一つ抜けるとごとの４点減点）これより短いものには、点は出さなかった。

問題５：

（１）　プロセスの実行状態を調べると

```
4 W    0   (-5)      0      0    97          ANY    ?   0:00 FLOPPY
0 R    0   (-4)      0      0    97                 ?   0:00 MEMORY
```

のように、FLOPPY という名称のタスクが常駐している。この名称で起動される関数は、どのファイルの何行目に定義されているか。ファイル名と行番号を記しなさい。（３点）

（２）　MEMORY という名称のタスクは、どのファイルの何行目に定義されているか。ファイル名と行番号を記しなさい。（３点）

（３）　FLOPPY も MEMORY も、これらのタスクが実際にデバイスドライバとして監視を行っている関数は、共通である。どのファイルの何行目に定義されている何という関数か。その関数の関数名、ファイル名と行番号を記しなさい。（３点）

（４）（３）の関数でデバイス監視を実際に行っている無限ループ構造（while 文）が存在しているファイル名と無限ループの先頭行番号を記しなさい。（５点）

※　MINIX のソースコードを編集して結果行番号が狂っている学生は、teachers_shared に置いてあるファイルを使用して下さい。（２~３行の誤差は、容認します。）

授業中何回か行った作業と、手順は全く同じ。/usr/src/fs と/usr/src/kernel の両方で FLOPPY が使われているが、”FLOPPY”が文字列として使われている側が答えになる。

まず、

grep FLOPPY *.c

と入力して、ファイルを探してみる。/usr/src/kernel の側で検索すると

```
table.c:        { floppy_task,          FLOP_STACK,     "FLOPPY"        },
```

という記述が見つかる。ここで、floppy_task に着目して、floppy_task を探すと、

floppy.c の 268 行目に、floppy_task が見つかる。

（１）の答え：floppy.c の 268 行目

MEMORY も同様。

grep MEMORY *.c で検索すると、mem_task が”MEMORY”として table で結び付けられている。

grep mem_task *.c で mem_task を検索し、vi memory.c で memory.c を開き、/mem_task と入力し文字列を検索すると、54 行目で定義されているのが見つかる。

（２）の答え：memory.c の 54 行目

ここで、それぞれのコードを見てみると

```
PUBLIC void floppy_task()
{
/* Initialize the floppy structure. */

  struct floppy *fp;

  for (fp = &floppy[0]; fp < &floppy[NR_DRIVES]; fp++) {
        fp->fl_curcyl = NO_CYL;
        fp->fl_density = NO_DENS;
        fp->fl_class = ~0;
  }

  put_irq_handler(&f_hook, FLOPPY_IRQ, f_handler);
  enable_irq(&f_hook);          /* ready for floppy interrupts */

  driver_task(&f_dtab);
}
```

```
PUBLIC void mem_task()
{
  m_init();
  driver_task(&m_dtab);
}
```

いずれも、最終的には driver_task をコールしている。

grep driver_task *.c で、driver_task を探すと、driver.c の 64 行目に定義されている。

（3）の答え： 関数は driver_task, ファイル driver.c の６４行目

この関数を見てみると

```
PUBLIC void driver_task(dp)
struct driver *dp;      /* Device dependent entry points. */
{
/* Main program of any device driver task. */

  int r, caller, proc_nr;
  message mess;

  init_buffer();        /* Get a DMA buffer. */


  /* Here is the main loop of the disk task.  It waits for a mess
   * it out, and sends a reply.
   */

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

        /* Wait for a request to read or write a disk block. */
        receive(ANY, &mess);
```

この７９行目で、while(TRUE)の無限ループ構造が定義されている。

（4）の答え： driver.c の 79 行目

ちょうど、返却していないレポートの内容と重なってしまった。(いじわるではありません。うっかりしていました。この回のレポートは提出率が高かったため、配点の調整用に最後まで手元に置いていました。)その点は失礼しましたが、たぶん、自分でレポートを解いた学生は、かなり覚えていてくれたものと思います。

/usr/src/kernel/table.c の 112 行目、113 行目を見つけた学生、惜しい、あと一息なので２点としました。

```
    { floppy_task,      FLOP_STACK,     "FLOPPY"        },
    { mem_task,         MEM_STACK,      "MEMORY"        },
```

問題６：

以下の設問のうち、いずれか１つを選択して答えなさい。解答では、どのキーワードに着目して、どのように調べたかも記しなさい。必要に応じて、コメントの翻訳なども書き加えなさい。（８点）

※ MINIX のソースコードを編集して結果行番号が狂っている学生は、teachers_shared に置いてあるファイルを使用して下さい。（２~３行の誤差は、容認します。）

（1） rtl8139.c は、Ethernet（LAN）カードのデバイスドライバファイルである。LAN のデータ受け渡しで多くの関数が使用している構造体は何か。構造体の名称と定義されているファイル名・行番号を記述しなさい。また、PCI（Peripheral Component Interconnect）の割込みレジスタ番号などがまとめて定義されているファイルはどれか。

問題文から、ファイル rtl8139.c を開き、先頭部分を読んでみると、かなり多数の関数が構造体として定義された型 re_t （struct re）を引数としている。

この構造体 struct re が定義されているのは、ファイル sb8139.c の 127 行目～173 行目である。

```
_PROTOTYPE( static void rl_init, (message *mp)                          );
_PROTOTYPE( static void rl_pci_conf, (void)                             );
_PROTOTYPE( static int rl_probe, (re_t *rep)                            );
_PROTOTYPE( static void rl_conf_hw, (re_t *rep)                         );
_PROTOTYPE( static void rl_init_buf, (re_t *rep)
_PROTOTYPE( static void rl_init_hw, (re_t *rep)                         );
_PROTOTYPE( static void rl_reset_hw, (re_t *rep)                        );
_PROTOTYPE( static void rl_confaddr, (re_t *rep)                        );
_PROTOTYPE( static void rl_rec_mode, (re_t *rep)                        );
_PROTOTYPE( static void rl_readv, (message *mp, int from_int,
                                                    int vectored)       );
_PROTOTYPE( static void rl_writev, (message *mp, int from_int,
                                                    int vectored)       );
_PROTOTYPE( static void rl_check_ints, (re_t *rep)                      );
_PROTOTYPE( static void rl_report_link, (re_t *rep)                     );
_PROTOTYPE( static void mii_print_techab, (U16_t techab)                );
_PROTOTYPE( static void mii_print_stat_speed, (U16_t stat,
                                                    U16_t extstat)      );
_PROTOTYPE( static void rl_clear_rx, (re_t *rep)                        );
_PROTOTYPE( static void rl_do_reset, (re_t *rep)                        );
_PROTOTYPE( static void rl_getstat, (message *mp)                       );
_PROTOTYPE( static void reply, (re_t *rep, int err, int may_block)      );
_PROTOTYPE( static void mess_reply, (message *req, message *reply)      );
```

また、PCI を　grep PCI *.h　と入力して検索すると、ファイル pci.h で PCI 関係の多数の名称が定義されているほか、20 行目では「割込み Line レジスタ」などのコメントを確認できるため、該当するファイルは PCI.H である。

（2）　Minix システムが、音源カード SB16（Sound Blaster 16）の DSP（Digital Signal Processor）をサポートする時、システムの初期化で音源カードの割り込みを設定している部分はどこか。ファイルの名称と行番号を記述しなさい。また、音源カードの割込み番号はいくつか、割り込み番号雄が定義されているかファイルの名称と、行番号を記述しなさい。なお、ファイルは kernel 内にある。（問題文に誤植アリ・・）

問題文で、音源カード SB16 の DSP とあることから、/usr/src/kernel/sb16_dsp.c を調べてみる。（ファイルの名称そのものがヒントである。）

以前演習課題で行った割込み処理関数の登録は、put_irq_handler()の関数で行っていた。このファイル内で put_irq_handler を検索すると、290 行目に

```
/* register interrupt vector and enable irq */
put_irq_handler(SB_IRQ, dsp_handler);
```

の記述を見つけた。つまり、/usr/src/kernel/sb16_dsp.c の 290 行目で割り込みを登録している。この SB_IRQ が「割込み番号」であるため、grep SB_IRQ *.h で検索すると、sb16.h の 10 行目で、５と定義している。

```
/* IRQ, base address and DMA channels */
#define SB_IRQ          5
```

ファイル　sb16_dsp.c を見つけたところまでで、３点とした。

タスクの初期化部分からプログラムをトレースしても、この割込み番号登録を見つけることはできる。

316 行目は、惜しいが、これは「割込みの発行」になっている。（オマケで１点）

（3）　Minix のコンソール（console）出力では、ESC（エスケープ）シーケンスと呼ばれる制御コードで文字の色を変えたり、文字をブリンクさせたり、下線を引いたりすることができる。この ESC

シーケンスを処理しているプログラムの関数は何か。ファイル名と行番号を答えなさい。また、下線を引く開始部分を処理しているプログラム行を探し、行番号を答えなさい。

演習の console 出力で、背景色を変えた時に、/usr/src/kernel/console.c を調べた。

この console.c のファイルを vi console.c で開き、/ESC と入力して ESC という文字列を検索すると、389 行目に parse_escape という関数が見つかり、この関数を調べると、do_escape という関数をコールしている。コメントに、サポートしている ESC シーケンスが書かれていた。

parse_escape は、ファイル/usr/src/kernel/console.c の 389 行目であり、

do_escape は、ファイル/usr/src/kernel/console.c の 449 行目である。

さらに、文字列 underline で検索すると、コメントとして 629 行目に UNDERLINE という文字が読める。

コメントから、629 行目で処理していると考えられる。

```
/*===========================================================================
 *                                parse_escape
 *===========================================================================
PRIVATE void parse_escape(cons, c)
register console_t *cons;       /* pointer to console struct */
char c;                         /* next character in escape sequence */
{
```

```
case 4:     /* UNDERLINE */
    if (color) {
            /* Change white to cyan, i.e. lose red
             */
            cons->c_attr = (cons->c_attr & 0xBBFF);
    } else {
            /* Set underline attribute */
            cons->c_attr = (cons->c_attr & 0x99FF);
    }
    break;
```

（4） Minix で、同時に実行可能なプログラムの数は、プロセス管理配列の大きさの制約から、比較的小さい値に制約されている。その値はいくつか？また、どのファイルで定義され、何と言う定義名でプログラムされているか。

演習の do_fork()で、プロセスの生成を調べた。この do_fork を読むとと、NR_PROCS という名称が出てきて、「プロセスを生成する fork の途中で、テーブルが溢れないように」と書かれているため、この NR_PROCS を検索する。NR_PROCS は、/usr/src/mm/mproc.h の 44 行目を見るとメモリのプロセス管理テーブルの配列の大きさであることが、わかる。

grep NR_PROCS *.h で、/usr/src/mm で調べても見つからなかったが、他の MINIX の重要な名称と同様に/usr/include/minix で検索すると、

/usr/include/minix/config.h の 34 行目で、３２に定義されていて、コメントは「ユーザプロセスのためのプロセステーブルのスロットの大きさ」と書かれていた。

答え：/usr/include/minix/config.h の３４行目で、NR_PROCS が３２と定義されている。つまり、ユーザが同時に起動可能なプロセスの数は、３２である。

このプロセス数は、カーネル側で調べると NR_TASKS が加算されて、システムの常駐プロセスの数も加えられている。問題文が「ユーザプロセス」とは指定していないため、こちらからでも構わないが、プロセスの生成を「fork」から説明したため、mm の側から調べることを想定して出題した。システムプロセスを加算した記述でも、無論正解とする。

（５）　Minix でファイルにアクセスする場合、ディスクへのアクセス回数を減らすためにキャッシュと呼ばれるデータ領域が存在する。キャッシュにデータが存在しているかどうかを判定している部分は、どこか。そのファイル名と行番号を示しなさい。

この問題の「問題４」で、get_block という関数が cache.c というファイルの中にあることを検索した。この関数の先頭数行のコメントを読んでみると、「要求されたブロックが、ブロックキャッシュに存在するかチェックする。」と書かれている。すなわち、このファイルで（第１段の）キャッシュを検索していることがわかる。さらにコメントを読み進むと 50 行目のコメントで、「ハッシュのチェーンを検索する、」と書かれている。従って、/usr/src/fs/cache.c の 55 行目の

```
if( dev != NO_DEV ){
```

で、キャッシュにデータが存在しているか判定している。