```c
/*********************************************************************
Copyright 2010-2017 K.C. Wang, <kwang@eecs.wsu.edu>
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
*********************************************************************/


typedef unsigned char   u8;
typedef unsigned short u16;
typedef unsigned int   u32;

#include "uio.c"
#include "crt0.c"
//#include "string.h"

int pid;
char line[64], pathname[32], i2[32], i3[32];
char *name[16], components[64];
int nk;
#define EOF -1
extern char cr;

void putchar(const char c){ }

int getc()
{
   int c, n;
   n = read(0, &c, 1);

   /*********************************************************************
   getc from KBD will NOT get 0 byte but reading file (after redirect 0
   to file) may get 0 byte ==> MUST return 2-byte -1 to differentiate.
   *********************************************************************/

   if (n==0 || c==4 || c==0 ) return EOF;

   return (c&0x7F);
}

// getline() does NOT show the input chars AND no cooking:
// for reditected inputs from a file which may contain '\b' chars

int getline(char *s)
{
  int c;
  char *cp = s;

  c = getc();

  while ((c != EOF) && (c != '\r') && (c != '\n')){
    *cp++ = c;
     c = getc();
```

```
  }
  if (c==EOF) return 0;

  *cp++ = c;          // a string with last char=\n or \r
  *cp = 0;
  //printf("getline: %s", s);
  return strlen(s);  // at least 1 because last char=\r or \n
}


// gets() show each input char AND cook input line

int gets(char *s)
{
  int c; char *cp, *cq, temp[128];

  cp = temp;    // get chars into temp[] first

  c = getc();
  while (c!= EOF && c != '\r' && c != '\n'){
    *cp++ = c;
    if (c=='\r'){
      mputc('n');
    }
    mputc(c);
    if (c == '\b'){ // handle \b key
      mputc(' ');
      mputc('\b');
    }
    c = getc();
  }
  mputc('\n'); mputc('\r');

  if (c==EOF) return 0;

  *cp = 0;

  // printf("temp=%s\n", temp);

  // cook line in temp[] into s
  cp = temp; cq = s;

  while (*cp){
    if (*cp == '\b'){
      if (cq > s)
          cq--;
      cp++;
      continue;
    }
    *cq++ = *cp++;
  }
  *cq = 0;

  //printf("s=%s\n", s);

  return strlen(s)+1;  // line=CR or \n only return 1
}


int getpid()
{
   return syscall(0,0,0);
```

```
}

int getppid()
{
   return syscall(1,0,0);
}

void chname()
{
    char s[64];
    prints("input new name : ");
    gets(s);
    printf("s=%s\n", s);
    syscall(2, s, 0);
}


int getpri()
{
  return syscall(3,0,0);
}

int chpri(int value)
{
  return syscall(4,value,0);
}

int getuid()
{
  return syscall(4,0,0);
}

int chuid(int uid, int gid)
{
  return syscall(5,uid, gid);
}

int tswitch()
{
  return syscall(6,0,0);
}


int fork()
{
   return syscall(10, 0, 0);
}

int exec(char *cmd_line)
{
   return syscall(11, cmd_line, 0);
}

int wait(int *status)
{
  return syscall(12, status, 0);
}

/******** vfork in us.s *******
int vfork()
{
  return syscall(19,0,0);
```

```
}
*************************/

int thread(int fn, int stack, int *ptr)
{
  //printf("fn=%x stack=%x ptr=%x *ptr=%d\n", fn, stack, ptr, *ptr);
  return syscall(14, fn, stack, ptr);
}

// 15-19: mutex for threads


int mutex_creat()
{
  return syscall(15, 0,0);
}

int mutex_lock(int *m)
{
  return syscall(16, m, 0);
}

int mutex_unlock(int *m)
{
  return syscall(17, m, 0);
}

int mutex_destroy(int *m)
{
  return syscall(18, m, 0);
}


int mkdir(char *name)
{
   return syscall(20, name, 0);
}

int rmdir(char *name)
{
   return syscall(21, name, 0);
}

int creat(char *filename)
{
    return syscall(22, filename, 30);
}

int link(char *oldfile, char *newfile)
{
  return syscall(23, oldfile, newfile,0);
}

int unlink(char *file)
{
  return syscall(24, file, 0);
}


int symlink(char *oldfile, char *newfile)
{
  return syscall(25, oldfile, newfile);
```

```
}

int readlink(char *file, char *linkname)
{
  return syscall(26, file, linkname, 0);
}

int chdir(char *name)
{
   return syscall(27, name, 0);
}

int getcwd(char *cwdname)
{
   return syscall(28, cwdname, 0);
}

int stat(char *filename, struct stat *sPtr)
{
   return syscall(29, filename, sPtr);
}

int fstat(int fd, char *sptr)
{
  return syscall(30,fd,sptr,0);
}


int open(char *file, int flag)
{
   return syscall(31, file, flag);

}

int close(int fd)
{
   return syscall(32, fd);
}

int lseek(int fd, u32 offset, int ww)
{
   return syscall(33, fd, (u32)offset, ww);
}

int read(int fd, char *buf, int nbytes)
{
    return syscall(34, fd, buf, nbytes);
}

int write(int fd, char *buf, int nbytes)
{
    return syscall(35, fd, buf, nbytes);
}

//KC's pipe
int pipe(int *pd)
{
    return syscall(36, pd, 0);
}

int chmod(char *file, u16 mode)
{
```

```
   return syscall(37, file, mode);
}

int chown(char *file, int uid)
{
   return syscall(38, file, uid);
}

int touch(char *filename)
{
   return syscall(39, filename, 0);
}

int settty(char *tty)// the original ucode.c is fixtty
{
   return syscall(40, tty, 0);
}

int gettty(char *tty)
{
   return syscall(41, tty, 0);
}


int dup(int fd)
{
   return syscall(42, fd, 0);
}

int dup2(int fd, int gd)
{
   return syscall(43, fd, gd);
}

int mount(char *dev, char **mpt)
{
  return syscall(45, dev, mpt);
}

int umount(char *dev)
{
  return syscall(46, dev);
}

int getSector(u32 sector, char *ubuf, u16 nsector)
{
  return syscall(47, sector, ubuf, nsector);
}

int do_cmd(int cmd, u16 value)
{
  return syscall(48, cmd, value);
}

int kill(int sig, int pid)
{
  return syscall(50, sig, pid);
}

int signal(int sig, int catcher)
{
  return syscall(51, sig, catcher);
```

```
}

int pause(int t)
{
  return syscall(52, t);
}

int itimer(int t)
{
  return syscall(53, t);
}

int send(char *msg, int pid)
{
  syscall(54, msg, pid);
}

int recv(char *msg)
{
  syscall(55,msg, 0);
}

int do_texit()
{
  int pid = getpid();
  printf("thread %d texit()\n", pid);
  texit(pid);
}

int tjoin(int n)
{
  return syscall(56, n, 0);
}

int texit(int v)
{
  syscall(57,v,0);
}

/*************************************************
int ps(char *y)
{
   return syscall(44,y,0);
}
// ********** CDROM syscalls *****************
int setcolor(int color)
{
  return syscall(59, color,0);
}
************************************/

int sync()
{
  return syscall(60, 0, 0);
}

int ups()
{
  return syscall(61, 0, 0);
}

int thinit()
```

```
{
  return syscall(62, 0, 0);
}

int sbrk()
{
  return syscall(63, 0, 0);
}

int page_out(int n)
{
  return syscall(64, n, 0);
}

int getphypage(int x, int y)
{
  return syscall(65, x, y);
}

int pagetable()
{
  return syscall(66, 0, 0);
}

int getcs()
{
  return syscall(67,0,0);
}

int exit(int value)
{
   return syscall(9, value, 0);
}

int pwd()
{
  char cwd[64];
  getcwd(cwd);
  printf("%s\n\r", cwd);
  return 0;
}


// nk = eatpat(line, name);

int eatpath(char *line, char *name[ ])
{
  int i, n; char *cp;

  n = 0;
  for (i=0; i<16; i++)
      name[i]=0;

  cp = line;
  while (*cp != 0){
       while (*cp == ' ')
              *cp++ = 0;
       if (*cp != 0)
           name[n++] = cp;
       while (*cp != ' ' && *cp != 0)
              cp++;
       if (*cp != 0)
```

```
            *cp = 0;
        else
            break;
        cp++;
  }

  /*
  for (i=0; i < n; i++){
      if (name[i]){
         prints(name[i]); prints("  ");
      }
  }
  prints("\n\r");
  */
  return n;
}

int strcasecmp(char *s1, char *s2)
{
  char *cp;

  char t1[64], t2[64];
  strcpy(t1, s1);
  strcpy(t2,s2);

  //printf("t1=%s  t2=%s  ", t1, t2);

  cp = t1;

  while(*cp){  // all to lower case
    if (('A' <= *cp) && (*cp <= 'Z')){
      *cp = *cp - 'A' + 'a';
    }
    cp++;
  }
  //printf("t1=%s  ", t1);
  cp = t2;
  while(*cp){  // all to upper case
    if (('A' <= *cp) && (*cp <= 'Z')){
      *cp = *cp - 'A' + 'a';
    }
    cp++;
  }
  //printf("t2=%s\n", t1, t2);
  return strcmp(t1, t2);
}
```