/\* crt0.c : main0(s) called from u.s, where s = oigianl command string

tokenlize s into char \*argv[ ] and call main(argc, argv).

token() breaks up a string into argc of tokens, pointed by argv[]

\*/

/\* #include "uinclude.h" \*/

int argc;

char \*argv[32];

void token(char \*line)

{

char \*cp;

cp = line;

argc = 0;

while (\*cp != 0){

while (\*cp == ' ') \*cp++ = 0;

if (\*cp != 0)

argv[argc++] = cp;

while (\*cp != ' ' && \*cp != 0) cp++;

if (\*cp != 0)

\*cp = 0;

else

break;

cp++;

}

argv[argc] = 0;

}

void showarg(int argc, char \*argv[ ])

{

int i;

printf("argc=%d ", argc);

for (i=0; i<argc; i++)

//printf("argv[%d]=%s ", i, argv[i]);

printf("%s ", argv[i]);

prints("\n");

}

// BEFORE: r0 was trashed in goUmode(), so had to rely on r1->string

// NOW: r0 is NOT trashed in goUmode() ==> should be r0 alone

void main0(char \*s)

{

if (s){

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

token(s);

}

//showarg(argc, argv);

main(argc, argv);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ucode.c file \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#define NULL 0

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 strtok(char\* str, char\* outstr, char delim, int occurrences) {

int index = 0 ;

int length = 0, found = 0;

if (!str) { return 0; }

if (!delim) { return 0; }

while(outstr[index] != 0) {

outstr[index] = 0;

index++;

}

index = 0;

while(str[index] && occurrences > 0) {

if (str[index] == delim) {

occurrences--;

}

index++;

}

if (!str[index]) {

return 0;

}

char name[256];

int j;

for (j = 0; j < 256; j++) {

name[j] = 0;

}

j = 0;

while (str[index] && str[index] != delim) {

name[j] = str[index];

index++;

j++;

}

name[j] = 0;

char\* cp = name;

j = 0;

while(cp[j]) {

outstr[j] = cp[j];

j++;

}

return 1;

}

int mygetc(int fd)

{

int c, n;

n = read(fd, &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);

}

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;

mputc(c);

if (c == '\b'){ // handle \b key

mputc(' ');

mputc('\b');

}

c = getc();

}

mputc('\r'); mputc('\n');

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 flag, int ptr)

{

return syscall(14, fn, stack, flag, 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)

{

if (fd==0)

nbytes = 1;

return syscall(34, fd, buf, nbytes);

}

int write(int fd, char \*buf, int nbytes)

{

return syscall(35, fd, buf, nbytes);

}

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)

{

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)

{

printf("signal go ");

syscall(51, sig, catcher);

printf("signal return\n");

}

int pause(int t)

{

return syscall(52, t);

}

int itimer(int t)

{

printf("itimer go ");

syscall(53, t);

printf("ucode: itimer return\n");

}

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);

// args: char\* and char\* list

int eatpath(char \*line, char \*name[ ]) {

int i, n; char \*cp;

n = 0;

// zero out array

for (i=0; i<16; i++) {

name[i]=0;

}

// set pointer to char\* arg

cp = line;

// while cp is not null

while (\*cp != 0) {

// zero out spaces

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);

}

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

io.c file of MTX

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

char space = ' ';

char \*ctable = "0123456789ABCDEF";

char cr = '\r';

int puts(const char \*s){ }

#define printk printf

int printf(char \*fmt,...);

typedef struct ext2\_dir\_entry\_2 {

u32 inode; /\* Inode number \*/

u16 rec\_len; /\* Directory entry length \*/

u8 name\_len; /\* Name length \*/

u8 file\_type;

char name[255]; /\* File name \*/

} DIR;

typedef struct stat {

u16 st\_dev; /\* major/minor device number \*/

u16 st\_ino; /\* i-node number \*/

u16 st\_mode; /\* file mode, protection bits, etc. \*/

u16 st\_nlink; /\* # links; TEMPORARY HACK: should be nlink\_t\*/

u16 st\_uid; /\* uid of the file's owner \*/

u16 st\_gid; /\* gid; TEMPORARY HACK: should be gid\_t \*/

u16 st\_rdev;

long st\_size; /\* file size \*/

long st\_atime; /\* time of last access \*/

long st\_mtime; // time of last modification

long st\_ctime; // time of creation

long st\_dtime;

long st\_date;

long st\_time;

} STAT;

// UNIX <fcntl.h> constants: <asm/fcntl.h> in Linux

#define O\_RDONLY 00

#define O\_WRONLY 01

#define O\_RDWR 02

#define O\_CREAT 0100 /\* not fcntl \*/

#define O\_TRUNC 01000 /\* not fcntl \*/

#define O\_APPEND 02000

/\*

#define O\_RDONLY 0

#define O\_WRONLY 1

#define O\_RDWR 2

#define O\_APPEND 3

\*/

#define EOF -1

#define exit mexit

int mputc(char c)

{

write(1, &c, 1);

if (c=='\n')

write(1, &cr, 1);

return 0;

}

void prints(char \*s)

{

while (\*s){

mputc(\*s);

s++;

}

}

void mputs(char \*s)

{

prints(s);

}

extern int strlen(const char \*);

void print2f(char \*s)

{

write(2, s, (int)strlen(s));

}

void rpi(int x)

{

char c;

if (x==0) return;

c = ctable[x%10];

rpi((int)x/10);

mputc(c);

}

void printi(int x)

{

if (x==0){

prints("0 ");

return;

}

if (x < 0){

mputc('-');

x = -x;

}

rpi((int)x);

mputc(space);

}

void rpu(u32 x)

{

char c;

if (x==0) return;

c = ctable[x%10];

rpi((u32)x/10);

mputc(c);

}

void printu(u32 x)

{

if (x==0){

prints("0 ");

return;

}

rpu((u32)x);

mputc(space);

}

void rpx(u32 x)

{

char c;

if (x==0) return;

c = ctable[x%16];

rpx((u32)x/16);

mputc(c);

}

void printx(u32 x)

{

prints("0x");

if (x==0){

prints("0 ");

return;

}

rpx((u32)x);

mputc(space);

}

void printc(char c)

{

c = c & 0x7F;

mputc(c);

}

int printk(char \*fmt, ...)

{

char \*cp;

int \*ip;

cp = (char \*)fmt;

ip = (int \*)&fmt + 1;

while (\*cp){

if (\*cp != '%'){

mputc(\*cp);

cp++;

continue;

}

cp++;

switch(\*cp){

case 'd' : printi(\*ip); break;

case 'u' : printu(\*ip); break;

case 'x' : printx(\*ip); break;

case 's' : prints((char \*)\*ip); break;

case 'c' : printc((char)\*ip); break;

}

cp++; ip++;

}

}

/\*\*\*\*\*\*\*\*\*\* test.c file \*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "ucode.c"

char buf[1024];

char\* names[5];

int main(int argc, char \*argv[ ]) {

struct stat mystat, \*sp;

int r;

char \*s;

char filename[1024], cwd[1024];

names[0] = ".";

names[1] = "..";

names[2] = "bin";

names[3] = "f1";

names[4] = NULL;

getcwd(cwd);

s = argv[1];

if (argc == 1) {

s = cwd;

}

sp = &mystat;

int i = 0;

while(names[i] != NULL) {

if ((r = stat(names[i],sp)) < 0) {

prints("cannot stat\n\r"); return -1;

}

ls\_entry(names[i], sp);

i++;

}

}

int ls\_entry(char\* name, struct stat \*sp){

printf("%x ", sp->st\_mode);

printf("%d ", sp->st\_nlink);

printf("%d ", sp->st\_uid);

printf("%d ", sp->st\_gid);

printf("%d ", sp->st\_size);

printf(" %s\n\r", name);

}

/\*\*\*\*\*\*\*\*\*\* init.c file \*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "ucode.c"

char buf[1024];

int console;

int s1;

int s0;

int in1, out1, in2, out2, in3, out3;

int parent() {

int pid, status;

while(1) {

prints("[INIT] Wait for a zombie child\n");

pid = wait(&status);

if (pid==console) { //if console died

console = fork();

if (console) { continue; }

else {

char\* line = "init - console login\n\r";

write(out1, line, 31);

exec("login /dev/tty0");

}

}

else if (pid == s0) {

s0 = fork();

if (s0) { continue; }

else {

char\* line = "init - primary port login\n\r";

write(out2, line, 26);

exec("login /dev/ttyS0");

}

}

else if (pid == s1) {

s1 = fork();

if (s1) { continue; }

else {

char\* line = "init - secondary port login\n\r";

write(out3, line, 26);

exec("login /dev/ttyS1");

}

}

printf("init - proc removed %d\n", pid);

}

}

int main(int argc, char \*argv[ ]) {

in1 = open("/dev/tty0", O\_RDONLY); // console read

out1 = open("/dev/tty0", O\_WRONLY); // console write

in2 = open("/dev/ttyS0", O\_RDONLY); // primary port read

out2 = open("/dev/ttyS0", O\_WRONLY); // primary port write

in3 = open("/dev/ttyS1", O\_RDONLY); // secondary port read

out3 = open("/dev/ttyS1", O\_WRONLY); // secondary port write

//fork console

int status;

console = fork();

while(1) {

if (console) {

s0 = fork(); //fork primary port

if (s0) {

s1 = fork(); //fork secondary port

if (s1) {

parent(); //wait for child proc to die

}

else {

char\* line = "Login on port: s1\n\r";

write(out3, line, 19);

exec("login /dev/ttyS1");

}

}

else {

char\* line = "Login on port: s0\n\r";

write(out2, line, 19);

exec("login /dev/ttyS0");

}

}

else {

char\* line = "Login on port: console\n\r";

write(out1, line, 24);

exec("login /dev/tty0");

}

}

return 0;

}

/\*\*\*\*\*\*\*\*\*\* login.c file \*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "ucode.c"

char buf[1024];

// upon entry, argv[0]=login, argv[1]=/dev/ttyX (virtual console)

int main(int argc, char \*argv[ ]) {

int i;

int in, out, err;

int passfd, n;

char name[128]; // stores username

char pass[128]; // stores password (from passwd file)

if (argc < 2) {

printf("ERROR - need at least 2 arguments\n");

printf("Sample usage: login /dev/tty0\n");

return -1;

}

//fd from init

close(0);

close(1);

in = open(argv[1], 0);

out = open(argv[1], 1);

err = open(argv[1], 2);

// set tty name in the procs

settty(argv[1]);

//confirm username and passwords

passfd = open("/etc/passwd", O\_RDONLY);

while (1) {

prints("username:\n\r"); gets(name);

prints("password:\n\r"); gets(pass);

n = read(passfd, buf, 1024);

if (n <= 0) {

prints("ERROR - no passwords were found.\n");

close(passfd); close(in); close(out); close(err);

return -1;

}

//token is username

int toki = 0;

char token[256];

int result = strtok(buf, token, '\n', toki++);

if (!result) { printf("nothing found!\n\r"); return -1; }

//look through every line in the password file

while (token) {

char token\_name[32];

strtok(token, token\_name, ':', 0);

if (!strcmp(token\_name,name)) {

char token\_pass[32];

strtok(token, token\_pass, ':', 1);

printf("pass = %s\n\r", token\_pass);

//password and password file have a match

if (!strcmp(token\_pass, pass)) {

char uid[16];

char gid[16];

char am[32];

char alias[16];

char mode[16];

char directory[16];

char execcmd[16];

strtok(token, uid, ':', 2);//username

strtok(token, gid, ':', 3);//password

strtok(token, am, ':', 4);

strtok(am, mode, ' ', 1);

strtok(am, alias, ' ', 0);

strtok(token, directory, ':', 5);

strtok(token, execcmd, ':',6);

//set username and password to users username and password

//set the current working directory to the user's home directory

//close password file

printf("uid=%s, gid=%s, alias=%s, mode=%s, dir=%s, cmd=%s\n",

uid, gid, alias, mode, directory, execcmd);

chuid(atoi(uid), atoi(gid)); //username and password

chdir(directory);

close(passfd);

printf("Welcome %s. Taking you to %s\n\r", token\_name, directory);

exec(execcmd);

return 1;

}

}

memset(token, 0, 32);

strtok(buf, token, 10, toki++);

if (token[0] == 0) {

printf("ERROR - login failed\n\r");

break;

close(passfd);

passfd = open("/etc/passwd", O\_RDONLY);

}

}

}

printf("exit\n");

}

char cwd[32];

char buf2[1024];

char \*t1 = "xwrxwrxwr-------";

char \*t2 = "-----------------";

int main(int argc, char \*argv[ ]) {

struct stat\* sbuf, mystat;

int n, r;

int fd;

int i;

sbuf = &mystat;

getcwd(cwd);

if (argc > 1) {

fd = open(argv[1], O\_RDONLY);

strcpy(cwd, argv[1]);

}

else {

fd = open(cwd, O\_RDONLY);

}

chdir(cwd);

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

if (fd < 0) {

printf("File not opened successfully\n");

close(fd);

return 0;

}

n = read(fd, buf2, 1024);

char\* cp = buf2;

DIR\* dp = (DIR\*)cp;

i = 0;

char copy[32];

while (cp < buf2 + 1024) {

memset(copy, 0, 32);

strncpy(copy, dp->name, dp->name\_len);

strcat(copy, "\0");

if ((r = stat(copy, sbuf)) < 0) {

printf("cannot stat %s\n\r", dp->name);

}

else {

ls\_entry(copy, sbuf);

}

cp += dp->rec\_len;

dp = (DIR\*)cp;

i++;

}

i = 0;

close(fd);

}

int ls\_entry(char\* name, struct stat \*sp){

if (sp->st\_mode == 0x41ed) {

mputc('d');

}

else {

mputc('-');

}

int i;

for (i = 8; i >= 0; i--) {

if (sp->st\_mode & (1<<i)) {

mputc(t1[i]);

}

else {

mputc(t2[i]);

}

}

printf(" %d ", sp->st\_nlink);

printf("%d ", sp->st\_uid);

printf("%d ", sp->st\_gid);

printf("%d ", sp->st\_size);

printf(" %s\n\r", name);

}

/\*\*\*\*\*\*\*\*\*\* grep.c file \*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "ucode.c"

char buf[1024];

char tty[32];

int main(int argc, char \*argv[ ]) {

prints("GREP\n\r");

int i, j, k, n, length;

int in, out;

char\* pattern;

char line[128];

char c;

gettty(tty);

int outtty = open(tty, O\_WRONLY);

if (argc < 2) {

prints("Not enough arguments\n\r");

return -1;

}

// STDIN

if (argc == 2) {

pattern = argv[1];

in = 0;

out = 1;

}

// FILE

else {

pattern = argv[1];

in = open(argv[2], O\_RDONLY);

if (in < 1) {

printf("Bad file: %s\n\r", argv[2]);

return -1;

}

out = 1;

}

//Check if it is a proper pattern

if (!pattern) { return -1; }

//caculate the length of the pattern

length = 0;

i = 0;

while (pattern[length] != '\0') {

length++;

}

i = j = k = 0;

int to\_print = 0;

memset(line, 0, 128);

while (1) {

n = read(in, buf, 1); //Read in from the buf one byte at a time

if (n < 1) { break; } // exit if we have run out of bytes

line[i] = buf[0];

if (buf[0] == 10) {

line[++i] = 13;

if (to\_print) {

write(out, line, i);

write(outtty, "\r", 1);

}

to\_print = 0;

i = j = 0;

memset(line, 0, 128);

}

else {

if (line[i] == pattern[j]) {

j++;

if (j == length && !to\_print) {

to\_print = 1;

}

}

else {

j = 0;

}

i++;

}

}

close(in); close(out);

}

/\*\*\*\*\*\*\*\*\*\* cat.c file \*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "ucode.c"

char buf[24];

char string[128];

char tty[32];

int main(int argc, char \*argv[ ]) {

int n;

int in, out;

int i;

gettty(tty);

int ttyfd = open(tty, O\_WRONLY);

//STDIN

if (argc == 1) {

in = 0;

out = 1;

}

// FILE

else {

in = open(argv[1], O\_RDONLY);

out = 1;

}

i = 0;

while(1) {

n = read(in, buf, 1); //read 1 byte from input

if (n < 1) { break; } //if we get nothing, break out

string[i] = buf[0];

if (!in) { write(ttyfd, buf, 1); }//cat without parameters

//new line

if (string[i] == '\n') {

i++;

string[i] = '\r';

write(out, string, i);

write(ttyfd, "\r", 1); //return cursor without stuffing the string

memset(string, 0, 128);

i = 0;

}

//"enter" key pressed, STDIN

else if (string[i] == '\r' && !in) {

string[i++] = '\n';

string[i++] = '\r';

write(ttyfd, "\n\r", 2); //return cursor without stuffing the string

write(out, string, i);

memset(string, 0, 128);

i = 0;

}

else {

i++;

}

}

close(in); close(out); close(ttyfd);

}

/\*\*\*\*\*\*\*\*\*\* more.c file \*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "ucode.c"

char buf[1024];

char buf2[1024];

int main(int argc, char \*argv[ ]) {

int fd, n;

int in, out;

char tty[32];

gettty(tty);

int outtty = open(tty, O\_WRONLY);

int intty = open(tty, O\_RDONLY);

int lines = 0;

int stopprint = 0;

write(outtty, "MORE\n\r", 14);

if (argc < 2) {

in = 0;

out = 1;

}

else {

in = open(argv[1], O\_RDONLY);

out = 1;

}

while ((n = read(in, buf, 1)) == 1) {//read in 1 byte at a time

if (!in) { write(outtty, buf, 1); }//spit it out if its stdin

else { write(out, buf, 1); }

if (buf[0] == '\n') {

lines++;

if (lines == 25) {

stopprint=1;

}

if (stopprint) {

while (1) {

char c = 0;

c = mygetc(intty);

if (c == '\n' || c == '\r') { break; }

else if (c == ' ') {

lines = 0;

stopprint = 0;

break;

}

}

}

prints("\r");

}

}

close(in); close(out); close(outtty);

}

/\*\*\*\*\*\*\*\*\*\* pipe.c file \*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "ucode.c"

char buf[1024];

char uinput[128];

char args[10][32];//tokenize pipe arguments

char ios[10][2];//io redirection

int child;

int main(int argc, char \*argv[ ]) {

int i;

int n;

int in, out;

int status;

char tty[32];

gettty(tty);

int stdin = 0;

while (1) {

for (i = 0; i < 10; i++) {

memset(args[i], 0, 32);

memset(ios[i], 0, 2);

}

i = 0;

printf("\n\r scsh #%d $ ", getpid());//receive input

gets(uinput);

if (uinput[0] == 0) {

continue;

}

if (!strcmp(uinput, "logout")) {

break;

}

while(strtok(uinput, args[i], '|', i)) { //tokenize pipes into arguments

i++;

}

int count = i;

i--;

printf("count = %d\n\r", count);

char\* cp;

int len;

//clear whitespace

for (; i >= 0; i--) {

cp = args[i];

len = 0;

while(\*cp == ' ') { cp++; }

strcpy(args[i], cp);

printf("args[%d] = %s\n\r", i, args[i]);

//check for io redirection

int j = 0;

while (args[i][j]) {

if (args[i][j] == '>') {

if (args[i][j+1] == '>') { //check if its append

strcpy(ios[i], ">>");

break;

}

else {

strcpy(ios[i], ">"); //check if its create

break;

}

}

else if (args[i][j] == '<') { //check if it takes input

strcpy(ios[i], "<");

break;

}

j++;

}

}

i = 0;

for (i = 0; i < 3; i++) {

printf("args[i] = %s\t", args[i]);

printf("ios[i] = %s\n\r", ios[i]);

}

i = 0;

child = fork();

if (child) {

wait(&status); // wait for child to die

}

else {

if (count == 2) {

prints("CREATING PIPE\n\r");

if (ios[0][0] != 0) {

handle\_IO(args[0], ios[0]);

}

if (ios[1][0] != 0) {

handle\_IO(args[1], ios[1]);

}

do\_pipe(args[0], args[1]); //recursive call

}

else if (count == 1) {

if (ios[0][0] != 0) {

handle\_IO(args[0], ios[0]);

}

exec(args[0]);

}

}

memset(uinput, 0, 128);

i = 0;

}

}

int handle\_IO(char\* arg, char\* IO) {

char cmd[32];

char file[32];

int type;

int fd;

int i=0;

memset(cmd, 0, 32);

memset(file, 0, 32);

while(arg[i]) {

if (arg[i] == '<' || arg[i] == '>') {

i++;

break;

}

else {

cmd[i] = arg[i];

}

i++;

}

int j = 0;

while (arg[i] == '>' || arg[i] == ' ') {

i++;

}

while (arg[i] && arg[i] != ' ') {

file[j] = arg[i];

j++;

i++;

}

printf("cmd = %s\n\r", cmd);

printf("file = %s\n\r", file);

strcpy(arg, cmd);

//IO redirection

if (!strcmp(IO, ">")) {

fd = open(file, O\_WRONLY|O\_CREAT); //open

if (fd<=0) { printf("ERROR - Couldn't open/create file %s\n\r", file); return -1; }

dup2(fd, 1);

//redirect stdout to file

close(fd);

exec(cmd);

return 1;

}

if (!strcmp(IO, ">>")) {

fd = open(file, O\_WRONLY|O\_CREAT|O\_APPEND);

if (fd<=0) { printf("Couldn't open/create file %s\n\r", file); return -1; }

dup2(fd, 1);

close(fd);

exec(cmd);

return 1;

}

if (!strcmp(IO, "<")) {

fd = open(file, O\_RDONLY); //unlike the others this one is read only

if (fd<=0) { printf("Couldn't open file %s\n\r", file); return -1; }

dup2(fd, 0); //redirect stdin instead

close(fd);

exec(cmd);

return 1;

}

}

int do\_pipe(char\* cmd1, char\* cmd2){

int pid, pd[2];

int status;

pipe(pd); //pd[0] = READ, pd[1] = WRITE

pid = fork();

if (pid) {

close(pd[1]);

dup2(pd[0], 0); //redirect stdin to pipe

exec(cmd2);

}

else {

close(pd[0]);

dup2(pd[1], 1); //redirect stdout to pipe

exec(cmd1);

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\* l2u.c file \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include "ucode.c"

char buf[1024];

char buf2[1024];

char string[128];

char tty[32];

char\* lower = "abcdefghijklmnopqrstuvwxyz";

char\* upper = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

int convert(char c) {

int i = 0;

while(lower[i]) {

if (lower[i] == c) {

return upper[i];

}

i++;

}

return -1;

}

int main(int argc, char \*argv[ ]) {

int n;

int in, out;

int i, j;

gettty(tty);

int outtty = open(tty, O\_WRONLY);

// STDIN

if (argc == 1) {

in = 0;

out = 1;

}

// FILE

else {

in = open(argv[1], O\_RDONLY);

out = 1;

}

i = j = 0;

char last;

while(1) {

n = read(in, buf, 1); //read input

if (n < 1) { break; } //bytes? break out

string[i] = buf[0];

int c = convert(buf[0]);

if (c != -1) {

string[i] = c;

buf[0] = c;

}

if (!in) { write(outtty, buf, 1); }

if (string[i] == 10) {

string[++i] = '\r';

write(out, string, i);

write(outtty, "\n\r", 1);

memset(string, 0, 128);

i = 0;

}

//enter key

else if (string[i] == 13 && !in) {

string[i++] = 10; //newline character

string[i++] = 13; //return character

write(outtty, "\r\n", 2);

write(out, string, i);

memset(string, 0, 128);

i = 0;

}

else {

i++;

}

}

close(in); close(out); close(outtty);

}

.global u\_entry, main0, syscall, getmysp, getcsr, getmyaddress

.global mexit

.text

.code 32

// upon entry, bl main0 => r0 contains pointer to the string in ustack

u\_entry:

mov r0, r1

bl main0

mov r0, #0

bl mexit

// if main0() ever retrun: syscall to exit(0)

@ user process issues int syscall(a,b,c,d) ==> a,b,c,d are in r0-r3

syscall:

// mov r4, sp // r4 = usp

// mov r5, pc // r5 = return PC

swi #0

mov pc, lr

getmysp:

mov r0, sp

mov pc, lr

getcsr:

mrs r0, cpsr

mov pc, lr

getmyaddress:

ldr r0, =main0

mov pc, lr

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

qemu with -kernel file.bin ==> start to execute from 0x10000 ==>

file.bin must be loaded to 0x10000 ==> . = 0x10000

\*\*\*\*\*\*\*\*\*\*\*\*/

ENTRY(u\_entry)

SECTIONS

{

. = 0x0;

.text : {

us.o

\*(.text)

}

.data : { \*(.data) }

.bss : { \*(.bss) }

}

if [ $# \< 1 ]; then

echo "mk FILENAME (without .c)"

exit

fi

arm-none-eabi-as -mcpu=arm926ej-s us.s -o us.o

arm-none-eabi-gcc -w -c -mcpu=arm926ej-s -o $1.o $1.c #2> /dev/null

arm-none-eabi-ld -T u.ld us.o $1.o eoslib -Ttext=0x80000000 -o $1

mount -o loop sdimage /mnt

cp -av $1 /mnt/bin

#ls -l /mnt/bin

umount /mnt

rm $1 \*.o

echo ready to go?

read dummy

qemu-system-arm -M versatilepb -m 256M -sd sdimage -kernel wanix \

-serial mon:stdio -serial /dev/pts/0