Shell-ul; cazul Unix

Mihai Budiu — mihaib+@cs.cmu.edu
`http://www.cs.cmu.edu/~mihaib`

11 aprilie 1997

Subiect: Ce este un shell; un exemplu pentru Unix.

Cunoştinţe necesare: Programare în C sub Unix de cel puţin un 1 an vechime.

Cuvinte cheie: proces, apel de sistem, shell, biblioteci, interpretor.

Contents

Orice utilizator al unui calculator interacţionează la un moment dat cu shell-ul. Fie că apare sub forma unor iconiţe mititele, fie sub forma unui *prompt* care invită la tastarea unor comenzi, shell-ul este prima interfaţă pe care un sistem de operare o oferă utilizatorilor săi. În acest articol ne propunem să arătăm că shell-ul este un program ca oricare altul şi să arătăm cum poate fi construit unul în cazul sistemului de operare Unix.

1 Ce este un shell?

Cuvîntul *shell* înseamnă în limba engleză *scoică* sau carapace. Asta este şi programul care se numeşte shell pentru sistemul de operare: un înveliş care îmbracă sistemul, primul lucru pe care îl vedem dintr-un sistem de operare, şi pe care trebuie să-l dăm la o parte pentru a zări măruntaiele moluştei.

Pentru a fi concreţi ne vom plasa în contextul sistemului de operare Unix. Ideile prezentate sunt însă valabile şi pentru alte sisteme de operare, deşi corespondenţa poate să nu fie evidentă.

Teza centrală a acestui articol este următoarea: un nucleu de sistem de operare este o bibliotecă de funcţii, care pune la dispoziţia utilizatorului o serie de operaţii, numite "apeluri de sistem" (vedeţi şi articolul "Nucleul" din PC Report Decembrie 1996?). Shell-ul este un simplu program care foloseşte aceste operaţii pentru a oferi o primă interfaţă cu utilizatorul.

```
               utilizator
                  |       ^
         comenzi|       |rezultate
        --------v-------|---------
        |    shell  |  proces    |
        |           |            |
        |---v----------------^---|
        |    \_creaza_proces_/   |
        |                        |
        |         nucleu         |
        --------------------------
        |       hardware         |
        --------------------------
```

1.1 Bucla principală

Misiunea unui shell este de a citi de la utilizator comenzi pe care apoi le interpretează şi le execută. Executarea comenzilor se face folosind apelurile de sistem. Pe scurt, orice shell are structura următoare:

```
while (1) {
/* 1 */   scrie_prompt();
/* 2 */   linie = citeste_linie_de_la_utilizator();
/* 3 */   comanda = identifica_comanda(linie);
/* 4 */   argumente = identifica_argumente(linie);
/* 5 */   executa_comanda(comanda, argumente);
/* 6 */   asteapta_terminarea_comenzii();
}
```

Apelurile de sistem sunt folosite pentru:

- A scrie utilizatorului mesaje (prompt);

- a citi tastele apăsate de utilizator;
- a ruga nucleul să execute o comandă;
- a aştepta executarea acesteia să se termine.

Observaţi că shell-ul *nu* scrie nici un fel de rezultate pe ecran (poate doar cînd s-a produs o eroare)! Shell-ul citeşte un nume de comandă, pe care o execută ca un proces separat. Acest proces este cel care tipăreşte rezultatele văzute de utilizator.

De exemplu, cînd folosiţi un sistem Unix, o interacţiune tipică este:

```
UNIX(r) System V Release 4.0 (hoho.cs.cornell.edu)

login: budiu
Password:
Last login: Thu Apr 10 17:42:44 from hyndla.cs.cornel
[budiu]:/home/budiu $ /bin/id
uid=831(budiu) gid=831(budiu)
[budiu]:/home/budiu $ /bin/ls /
cdrom       etc         lib         usr         export      lost+found
var         mnt         proc        bin         dev         home
net         sbin        devices     kernel      tmp
```

Textul `login:` este scris de programul `getty`, care aşteaptă să vină cineva la terminal. Textul `Password:` este scris de programul `login`, care vrea să mă identifice. După ce am fost identificat, porneşte shell-ul. (Paragraful acesta se poate dovedi eliptic; vom reveni poate cu altă ocazie despre sistemul de autentificare din Unix; vedeţi şi articolul din PC Report "Uşa din spate" din aprilie 1997 pentru informaţii).

În fine, textul `[budiu]:/home/budiu $` este scris de shell-ul meu. Acesta este *prompt*-ul, prin care shell-ul mă invită să tastez o comandă pentru el. Dacă ne referim la pseudo-codul de mai sus, (din secţiunea "bucla principală") acum s-a executat linia `/* 1 */`.

Eu tastez întîi `/bin/id` ca să văd cine sunt eu. Shell-ul execută linia `/* 2 */`, captînd caracterele apăsate de mine şi strîngîndu-le la un loc.

Cînd tastez ENTER, shell-ul pricepe că am terminat o comandă, şi trece la liniile `/* 3 */` şi apoi `/* 4 */`. Descifrează cuvîntul `/bin/id`, care este numele unui fişier. Acest fişier este codul executabil al comenzii `id`, pe care eu o doresc executată. Pentru că nu mai există alte cuvinte după `/bin/id`, această comandă nu are argumente. (Sintaxa comenzilor este o convenţie impusă de cel care a scris shell-ul).

Acum shell-ul meu trece la faza `/* 5 */`, rugînd nucleul sistemului de operare să execute fişierul `/bin/id`. Shell-ul între timp aşteaptă terminarea ei, în faza `/* 6 */`. Comanda `/bin/id` este executată şi provoacă scrierea pe ecran a textului `uid=831(budiu) gid=831(budiu)`.

Pentru că `id` s-a terminat, shell-ul o ia de la capăt în bucla `while`, citind o nouă comandă.

Pentru că voi tasta de data asta `/bin/ls /`, shell-ul va interpreta asta ca o comandă `/bin/ls` urmată de un argument, directorul rădăcină, `/`, şi va ruga nucleul să execute comanda `ls` cu argumentul `/`. Rezultatul executării se vede pe ecran: `ls` scrie conţinutul directorului rădăcină.

Observaţi cum pe ecran alternează texte scrise de shell şi de comenzile executate de el.

2 Cel mai simplu exemplu

Şi ca să nu rămînem în abstract, iată probabil cel mai simplu exemplu posibil de shell pentru sistemul Unix, care este pe deplin funcţional! (Încercaţi-l!) Shell-urile moderne oferă o grămadă de funcţionalităţi suplimentare care fac viaţă utilizatorului mai simplă, dar la rigoare exemplul din codul de mai jos este perfect suficient, deşi nu întotdeauna foarte convenabil de folosit. O virtute incontestabilă însă are, şi anume simplitatea, lucru care îl şi face subiectul nostru de studiu.

Acest program este luat de pe Internet, din grupul de discuţii `comp.unix.shell`, unde eu l-am văzut trimis de Brian S. Hiles (bsh20858@news.fhda.edu).

```
/* smallest shell */
a,b[99],*c,d[99];main(){while(printf(">"),c=d,*c=a=gets(b)){
for(;*++c=strtok(a," ");a=0);fork()?wait(0):execvp(*d,d+1);}}
```

Puteţi compila şi executa acest program pe orice platformă Unix care posedă un compilator de C. Dacă fişierul se numeşte `shell.c`, puteţi sa o faceţi în următorul fel:

```
$ cc shell.c -o shell
"shell.c", line 2: warning: old-style declaration or incorrect type for: a
"shell.c", line 2: warning: old-style declaration or incorrect type for: b
"shell.c", line 2: warning: old-style declaration or incorrect type for: c
"shell.c", line 2: warning: old-style declaration or incorrect type for: d
$ ./shell
>
```

(Nu trebuie să vă îngrijoreze eventuale *warning*-uri; programul este corect.)

Semnul `>` este prompt-ul. Puteţi tasta orice comandă:

```
> /bin/ls /
cdrom       etc         lib         usr         export      lost+found
var         mnt         proc        bin         dev         home
net         sbin        devices     kernel      tmp
> /bin/id
uid=831(budiu) gid=831(budiu)
>
```

Puteţi tasta CONTROL-d pentru a termina interacţiunea cu el.

Să vedem cum funcţionează această miniatură. Dacă aţi înţeles totul sunteţi un expert, atît în C, cît şi în Unix. Este remarcabil cîte tehnici interesante de programare se află într-un program atît de scurt...

2.1 Cărămizile: funcţiile de bibliotecă

Înainte de a încerca să înţelegem cum procedează, să analizăm fiecare din funcţiile C folosite. O parte din funcţii sunt C standard, si este oferită de orice compilator civilizat (`printf(), gets(), strtok()`), altele sunt apeluri de sistem Unix (`fork(), execvp(), wait()`).

2.1.1 printf()

Prima şi cea mai simplă funcţie folosită este `printf()`. Ea este folosită pentru a tipări prompt-ul. Această funcţie este o funcţie de bibliotecă, dar care este implementată la rîndul ei în termenii apelului de sistem `write()`, cu care în Unix se pot trimite date spre un periferic (în acest caz ecranul).

Deşi este extrem de interesantă în sine, nu-i vom consacra lui `printf()` mai multe rînduri, ca să nu ne îndepărtăm de scopul nostru declarat.

2.1.2 gets()

Funcţia `gets()` este "opusa" lui `printf()`: este folosită pentru a citi o linie de la utilizator. Argumentul ei este un array de caractere, unde va pune rezultatul citirii (un şir de caractere). Rezultatul apelului funcţiei este chiar argumentul ei (sau 0 dacă s-a întîlnit "sfîrşitul de fişier", ceea ce în cazul utilizatorului în general înseamnă că acesta a apăsat CONTROL-d). În plus, `gets()` schimbă sfîrşitul de linie (`\n`) cu un sfîrşit de şir (`\0`).

Ca şi `printf()`, `gets()` este o construită din cărămizi mai mici, din care principala este apelul de sistem `read()`, care (opusul lui `write()`), citeşte date de la un periferic sau fişier.

2.1.3 strtok()

Exotica funcţie `strtok()` este standard C, deşi probabil doar programatorii experimentaţi ştiu de ea. Declaraţia ei se găseşte în headerul `<string.h>`. Numele ei înseamnă "string tokenize", adică "împarte şir în cuvinte". Funcţionarea ei este relativ complexă.

Ea are două argumente, ambele şiruri de caractere, şi întoarce un pointer spre un şir de caractere:

```
char *strtok(char * text, const char * separatori);
```

Funcţionarea ei este specială dacă `text == NULL`. Scopul ei este de a împărţi şirul `text` în bucăţele care sunt despărţite de caractere din şirul `separatori`. Ca să înţelegem mai bine, iată întîi un exemplu care o foloseşte într-un mod tipic.

```
#include <string.h>
#include <stdio.h>
main()
{
   char * text = "sir  de separat   in bucati";
   char * separator = " ";
   char * cuvint;

   cuvint = strtok(text, separator);
   while (cuvint != NULL) {
        printf("%s\n", cuvint);
        cuvint = strtok(NULL, separator);
   }
}
```

Dacă executaţi acest program veţi obţine:

```
sir
de
separat
in
bucati
```

Prima oară cînd chem `strtok(text, separator)`, aceasta observă că `text` nu e 0 (`NULL`), deci îl memorează undeva într-un buffer intern funcţiei. După aceasta caută prima apariţie a unui caracter din `separator`, pe care o înlocuieşte cu un caracter 0 (sau `\0`), care în C înseamnă "sfîrşit de şir".

Lucrurile arată cam aşa la intrare:

```
text = |s|i|r| | |d|e| |s|e|p|a|r|a|t| | | |i|n| |b|u|c|a|t|i|\0|
       ----------------------------------------------------------
```

şi cam aşa la ieşire:

```
text = |s|i|r|\0| |d|e| |s|e|p|a|r|a|t| | | |i|n| |b|u|c|a|t|i|\0|
       -----------------------------------------------------------
        ^          ^

        cuvint     am ramas aici
```

Cu alte cuvinte şirul `text` a fost modificat, rezultatul `cuvint` punctează la primul cuvînt, care a fost separat cu un 0 de restul şirului, iar `strtok()` ţine minte unde a rămas.

A doua oară cînd îl chem pe `strtok()` îi dau un argument 0 în locul lui `text`. Asta înseamnă: "continuă de unde ai rămas". Aşa că rezultatul va fi:

```
text = |s|i|r|\0| |d|e|\0|s|e|p|a|r|a|t| | | |i|n| |b|u|c|a|t|i|\0|
       ------------------------------------------------------------
                   ^      ^

              cuvint      am ramas aici
```

Data viitoare voi obţine:

```
text = |s|i|r|\0| |d|e|\0|s|e|p|a|r|a|t|\0| | |i|n| |b|u|c|a|t|i|\0|
       -------------------------------------------------------------
                          ^                    ^

                          cuvint               am ramas aici
```

După încă două apeluri voi obţine rezultat 0, pentru că s-a ajuns la sfîrşitul şirului:

```
text = |s|i|r|\0| |d|e|\0|s|e|p|a|r|a|t|\0| | |i|n|\0|b|u|c|a|t|i|\0|
       --------------------------------------------------------------
                                                                 ^

cuvint=0                                                 am ramas aici
```

Iată deci cum `strtok()` hăcuieşte un şir de caractere. `strtok()` este o funcţie care poate fi implementată fără ajutorul vreunui apel de sistem, pentru că manipulează simple array-uri de caractere.

2.2 Apeluri de sistem

Ultimele 3 funcţii sunt apeluri de sistem tipice Unix, care au de-a face cu crearea de noi procese.

2.2.1 `fork()`

`fork()` este singura metodă prin care în Unix se poate crea un nou proces (vă reamintesc că un proces este un program în curs de execuţie). Cînd un proces execută `fork()`, el dă naştere unui alt proces absolut identic cu el însuşi, care se numeşte "fiul" lui, şi care îşi continuă execuţia în paralel cu tatăl.[1]

Amîndouă procesele îşi continuă execuţia ca şi cum tocmai ar fi executat instrucţiunea `fork()`; singura diferenţă dintre ele este că tatăl va primi ca rezultat al executării lui `fork()` un număr nenul, care identifică fiul, iar fiul va primi un 0.

Figura explică şi numele: "fork" înseamnă "furculiţă".

```
               --------------
               |            | proces=12
inainte        | r = fork() |
de             |            |
fork()         -------------- -----------
                     |                  \
                     v                   v
               --------------            --------------
               |            | proces=12  |            | proces=13
dupa           | r = fork() | (tata)     | r = fork() | (fiu)
fork()         | (r=13)     |            | (r=0)      |
               --------------            --------------
```

În figura de mai sus, părintele (cu numărul de proces 12) crează un copil, care întîmplător capătă numărul 13. După instrucţiunea `fork()` avem 2 procese, 12 şi 13, care se execută în paralel. Cel cu numărul 12 (părintele), va avea `r=13`, numărul copilului, iar copilul, deşi va executa acelaşi cod, din acelaşi punct, va avea `r=0`.

2.2.2 `wait()`

Cu apelul de sistem `wait()` un proces părinte aşteaptă terminarea unui copil. Părintele este blocat din execuţie pînă unul dintre copii lui răposează, după care `wait()` returnează părintelui nişte informaţii despre cauza decesului (nu ne preocupă aici). (Argumentul lui `wait()` este un pointer spre un loc unde se vor pune aceste informaţii; in programul nostru pointerul este 0; rezultatul lui `wait()` este numărul de proces copilului.)

2.2.3 `execvp()`

Rezultatul compilării unui program este un fişier executabil (de exemplu, mai sus cînd aţi compilat `shell.c` aţi obţinut `shell`). Apelul de sistem `exec()` roagă nucleul să citească un fişier executabil

[1] În Statele Unite această terminologie ("father, son") este descurajată, pentru că face discriminare sexuală. Se recomandă folosirea termenilor "parent" and "child". Ar trebui deci să spunem "părinte" şi "copil".

în memorie, să-l transforme în imaginea unui proces şi să-l execute, furnizînd şi argumentele care vor fi pasate noului proces. Procesul care execută apelul acesta de sistem este complet înlocuit cu imaginea din fişier.

Există mai multe variante ale apelului de sistem `exec()` care fac cam acelaşi lucru, dar care împachetează altfel argumentele. Cea de faţă se numeşte *execvp* pentru că este "exec" cu un: "Vector de Pointeri" spre argumentele procesului. Declaraţia din `<unistd.h>` arată aşa:

```
int execvp(const char *fisier, char * const argumente[]);
```

Primul parametru este numele fişierului executabil.

Al doilea parametru este un array în care fiecare obiect este un pointer la un argument pasat procesului. Argumentele sunt simple şiruri de caractere. Argumentele sunt primite de funcţia `main()` a procesului exec-utat în parametrul `argv`:

```
int main(int argc, char * argv[])
                          ^^^^^^

                         argumentele transmise de exec
```

Acum ştim toate bucăţile. Poate doar anumite caracteristici mai stranii ale C-ului mai stau în calea înţelegerii întregului program.

2.3 Ansamblul

Iată încă odată programul de mai sus, cu headerele incluse, indentat şi comentat:

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

int a,                          /* char * plimbat pe sirul citit */
    b[99],                      /* comanda tastata; folosit ca un char [] */
    *c,                         /* pointer spre argumentul curent */
    d[99];                      /* pointeri spre lista de argumente */

int main()
{
  while(printf(">"),            /* prompt */
        c=d,                    /* argumentele vor fi plasate in vectorul d */
        *c=a=gets(b)) {         /* citeste prima linie; a = b */
    for( ;
        *++c=strtok(a," ") ;    /* separa urmatorul cuvint; pune-l in d */
        a=0);                   /* a = NULL -> folosim acelasi buffer */
    fork() ?                    /* eu sunt tatal? */
      wait(0) :                 /* da, asteapta copilul! */
      execvp(*d,d+1);           /* nu: executa comanda data cu argumente */
}}
```

În primul rînd programul foloseşte numere întregi pentru a memora atît caractere cît şi pointeri. Standardul C spune că asta e perfect legal: "există un tip întreg în care un pointer încape".

Operatorul `,` este folosit cum scrie la carte: "evaluarea expresiei `e1,e2` înseamnă că:

- se evaluează `e1`;
- se evaluează *după aceea* `e2`;
- rezultatul final este rezultatul evaluării lui `e2`".

Variabila `a` punctează la şirul citit de la utilizator, pentru a-l împărţi în felii cu `strtok()`. Variabila `b` conţine chiar comanda tastată (e folosită ca un array de caractere). Variabila `d` va conţine la sfîrşitul executării lui `strtok()` pointerii către argumente daţi lui `execvp()`. În fine, variabila `c` se plimbă printre căsuţele lui `d`, arătînd care vine la rînd.

Situaţia la început arată astfel:

```
| | | | | | | | | | | | | ....  | b    | | | | | | | | | | | | | ....  | d
--------------------------------      --------------------------------
 ^                                     ^
 a                                     c
```

Să presupunem că utilizatorul tastează la prompt textul `echo a1 a2`. Atunci după executarea `for`-ului situaţia va fi următoarea:

```
|e|c|h|o|\0|a|1|\0|a|2|\0| | | | ....  |  b
---------------------------------------
 ^          ^      ^
 |  ________|      |
 | |  _____________/
 | | |
|*|*|*|0| | | | | | | | | | ....  |  d
------------------------------------
       ^
       c
```

Din cauza asta, după ce copilul e făcut şi părintele doarme, copilul execută comanda indicată de primul parametru al lui `execvp()`, `*d`, (care punctează spre şirul `echo`), iar argumentele lui `echo` sunt cuprinse în vectorul `d`, începînd de la căsuţa 1 (`d+1`).

3 Recompensă

Dacă aţi înţeles totul, vă propun spre "dezlegare" încă un cod de shell, de data asta mai sofisticat. Acesta poate recunoaşte mai multe comenzi pe aceeaşi linie, pipe-uri (`comanda1 | comanda2`), redirectări în fişiere (`comanda >fisier1 <fisier2`), comanda internă `cd`. Programul este perfect funcţional!

Pentru că sunt convins că nu veţi putea tasta o asemenea oroare, fişierul cu sursa este disponibil în pagina mea de web (`http://www.cs.cmu.edu/~mihaib/articles/`); luaţi-l şi compilaţi-l. Poate cu sprijinul redacţiei instaurăm şi un premiu pentru cel care reuşeşte să decodifice şi să comenteze programul într-un fel care să-l facă inteligibil. (Eu am reuşit parţial abia după două zile de lucru.)

```
#define D ,close(

char              *c,q              [512              ],m[              256
],*v[           99], **u,        *i[3];int         f[2],p;main       (){for
 (m[m        [60]=   m[62      ]=32   ]=m[*      m=124   [m]=       9]=6;
  e(-8)     ,gets      (1+(    c=q)     )||      exit      (0);     r(0,0)
   )for(    ;*++        c;);  }r(t,      o){    *i=i         [2]=   0;for
     (u=v  +98           ;m[*--c]         ^9;m [*c]          &32  ?i[*c
       &2]=                *u,u-            v^98              &&++u:

        3       )if(!m[*c]){for(*++c=0;!m[*--c];);
        *       --u= ++c;}u-v^98?strcmp(*u,"cd")?*c?pipe(f),o=f[
        1       ]:
        4       ,(p=fork())?e(p),o?r(o,0)D o)D*f):
        1       ,wait(0):(o?dup2(*f,0)D*f)D o):*i?
        5       D 0),e(open(*i,0)):
        9       ,t?dup2(t,1)D t):i[
        2       ]?
        6       D 1),e(creat(i[2],438)):
        5       ,e(execvp(*u,u))):e(chdir(u[1])*2):
        3       ;}e(x){x<0?write(2,"?\n$ "-x/4,2),x+1||exit(1):
        5       ;}

/* from  bsh20858@news.fhda.edu (Brian S Hiles) */
```

4 Rezumat

Observaţi că, pentru nucleu, shell-ul este un proces ca toate celelalte, care se execută fără nici un fel de privilegii şi care ocazional face cîte un apel de sistem. Shell-urile citesc comenzi de la utilizator, pe care apoi le "interpretează" şi le transformă într-o serie de apeluri de sistem, care în general culminează cu executarea unuia sau mai multor fişiere.

Limbajul C este de o conciziune şi expresivitate admirabilă. Nu-i aşa?