C și LINUX în Finlanda

Mihai Pătrașcu

Începând de anul acesta, sistemul de operare folosit la olimpiadele internaționale de informatică va fi Linux. Limbajul C s-a impus de mult timp în lumea Unix și este, fără nici un dubiu, cel mai folosit limbaj sub aceste sisteme. Vom încerca să prezentăm câteva elemente pentru a ajuta tranziția programatorilor în C de la mediile Borland la uneltele GNU/Linux.

Mediul de programare

Varietatea de editoare și medii de programare existente ne împiedică să recomandăm unul anume. Fiecare programator are un mediu preferat și va găsi numeroase motive pentru a-și motiva alegerea. Personal, autorul folosește de mai mulți ani editorul `joe` și este foarte mulțumit de performanțele lui.

Puține editoare oferă și facilități de compilare a programelor, deci ar putea fi util să vă obișnuiți cu folosirea mai multor terminale. Deschizând două terminale, puteți folosi editorul dumneavoastră favorit în unul dintre acestea și *shell*-ul *bash* în cel de-al doilea, care va fi folosit pentru compilarea, rularea si depanarea programelor.

Să presupunem că ați creat fișierul **`sum.c`** care conține codul:

```
#include <stdio.h>
int main(){
  int a,b;
  fscanf(fopen("sum.in","r"),"%d%d",&a,&b);
  fprintf(fopen("sum.out","w"),"%d\n",a+b);
  return(0);
}
```

Puteți compila și testa acest program folosind comenzile:

```
% gcc sum.c -o sum -Wall -O2
% echo 3 7 > sum.in
% time sum
...      (timpii de rulare ai programului)
% cat sum.out
10
```

Caracterul `%` este *prompt*-ul afișat de *shell* și poate varia. La unele distribuții directorul curent nu este inclus în variabila de mediu `PATH` și este necesar să tastăm `time ./sum` pentru a rula programul. Sunt raportați trei timpi de rulare: ***timpul real*** (*wall-clock* time), ***timpul efectiv*** de rulare al programului (fiind vorba de un sistem *multitasking*, pot fi rulate și alte programe simultan, deci acest timp este mai mic decât cel real), și ***timpul logic*** care reprezintă timpul consumat efectiv de program, la care se adaugă timpul consumat de sistemul de operare pentru a efectua operații cerute de programul respectiv (de exemplu, pentru a scrie rezultatele pe disc). Pe un sistem pe care nu rulează nici un alt program, timpul fizic este egal cu cel logic (aceasta va fi probabil situația în timpul evaluării soluțiilor prezentate de concurenți la *IOI 2001*).

O greșeală frecventă la începători este să denumească sursa **`test.c`** și să creeze executabilul **`test`**. Încercarea de a rula acest executabil va da rezultate neașteptate pentru că există deja un utilitar *shell* cu acest nume și acesta va fi executat datorită ordinii directoarelor în `PATH`.

Pentru compilarea programelor se recomandă specificarea opțiunilor `-Wall -O2`, pentru afișarea tuturor avertismentelor generate de compilator și optimizarea codului generat. Dacă omitem opțiunea `-O2`, pe lângă generarea unui cod mai puțin eficient, mai există un efect secundar nedorit: unele posibile greșeli nu vor fi depistate și raportate, deși am specificat `-Wall`.

Programarea *shell* se poate dovedi a fi un atu în unele cazuri. Acesta este însă un domeniu prea vast pentru a-l descrie amănunțit în cadrul acestui articol. Ne vom opri totuși asupra unui utilitar: `bc`. Acesta oferă un limbaj script asemănător limbajului C, cea mai importantă facilitate fiind reprezentată de faptul că întregii pot fi oricât de mari, mărimea lor fiind limitată doar de memoria disponibilă. Acest utilitar poate fi un ajutor neprețuit în testarea programelor care folosesc numere mari. Să presupunem că dorim să calculăm valoarea `100!`. Cea mai comodă metodă este să tastăm:

```
% bc
for(fact=i=1;i<=100;i++) fact*=i;
```

```
fact
...       (bc afișează valoarea 100!)
quit
```

Observați că afișarea valorii unei expresii, în particular o variabilă, se face scriind, pur și simplu, acea expresie.

Depanarea

Puține programe funcționează perfect la prima rulare. Mai devreme sau mai târziu va trebui să depanăm un program care funcționează conform așteptărilor. Deși există unele medii de dezvoltare care au unele facilități de depanare, vom constata că există soluții foarte bune și fără a recurge la acestea.

Cea mai simplă și mai folosită metodă de depanare este modificarea programului pentru a afișa unele informații esențiale în anumite puncte critice. Astfel, citind ieșirea programului vom putea analiza evoluția sa. Dublată de puțină experiență, această abordare se dovedește foarte eficientă.

Deși există posibilitatea afișării informațiilor de depanare la ieșirea standard, este mult mai recomandabil să se folosească ieșirea de eroare pentru acest lucru (`stderr`). Un motiv este că cel mai comod mod de a lucra cu fișiere în timpul concursurilor este să redirectăm intrarea și ieșirea standard la fișierele specificate de problemă. Multe soluții conțin o secvență asemănătoare cu următoarea:

```
freopen("prob2.in","r",stdin);
scanf("%d",&n);
...
freopen("prob2.out","w",stdout);
printf("%d\n",min);
...
```

Un motiv mult mai puternic este că în cazul terminării subite a programului dumneavoastră (cum ar fi accesul la o adresă de memorie invalidă), mesajele scrise la ieșirea standard pot să nu fie trimise sistemului de operare și nu vor fi niciodată afișate.

Pentru a micșora numărul de caractere tastate pentru afișarea mesajelor la `stderr`, ne putem folosi de o extensie a compilatorului `gcc`:

```
#define pe(fmt,args...) \
fprintf(stderr, fmt ,##args)
...
pe("<i=%d,min=%d>",i,min);
```

Trebuie să fiți atenți la dispunerea spațiilor în jurul virgulei care precede cele două diezuri.

O altă variantă des folosită, care este conformă standardului *ANSI C*, este următoarea:

```
#define pr(x) fprintf(stderr, #x " = %d\n", x)
...
pr(n * k - i);
```

Presupunând că valoarea expresiei este 13, programul va scrie la ieșirea de eroare:

```
n * k - i = 13
```

O problemă o constituie programele care scriu o cantitate foarte mare de informații la ieșirea de eroare (uneori și compilatorul `gcc` generează sute de erori). În mod normal, un program de paginare ca `less` nu citește decât ieșirea standard. Îl putem face să citească și ieșirea de eroare prin următoarea secvență:

```
$ prob3 2>/dev/stdout | less
```

Standardul *ANSI C* prevede două facilități care ajută la depanare. Funcția `abort()` încheie execuția programului și generează un fișier *core*. Puteți apela această funcție când depistați un caz "imposibil". Astfel, când se întâmplă un eveniment neprevăzut, programul dumneavoastră se va opri și veți putea examina starea memoriei în acel moment, putând constata unde ați greșit în raționamentul dumneavoastră. Celălalt ajutor în depanare este funcția `assert()`. Acesta primește un parametru, de obicei o condiție care sunteți siguri că este îndeplinită. Dacă condiția nu este îndeplinită, se comportă asemănător cu `abort()`.

Dacă aveți experiență în programarea sub Linux, probabil vi s-a întâmplat ca programul dumneavoastră să se oprească și să fie generat un fișier *core*. Cea mai frecventă cauză este semnalul `SIGSEGV`, care este generat de încercarea programului de a accesa o zonă invalidă de memorie: accesarea unui pointer cu valoarea `NULL`, folosirea unui pointer neinițializat (care conține o valoare aleatoare) sau accesarea unui element de tablou folosind un indice invalid.

Să considerăm următorul exemplu:

```
% prob3
Segmentation fault (core dumped)
% gdb prob3 core
Program terminated with signal 11,
Segmentation fault.
#0  0x804839e in main () at prob3.c:7
7                 a[x[i] - k] = 1;
(gdb) print i
$1: 17
(gdb) print x[i]-k
$2: 17774212
```

Este evident că vectorul `a[]` a fost indexat cu o valoare incorectă. Mai trebuie precizat că unele distribuții setează o opțiune care împiedică generarea fișierelor *core*. În acest caz puteți folosi următoarea comandă:

```
$ ulimit -c 65536
```

Am făcut între timp cunoștință cu depanatorul *GNU*: `gdb`. În afara examinării fișierelor *core*, este foarte comodă folosirea depanatorului pentru examinarea cursului execuției programului și a evoluției variabilelor pentru cazuri de complexitate redusă. Să presupunem că, pentru a fi siguri de corectitudinea programului, ne interesează valorile unui vector după execuția unei anumite funcții. Cea mai rapidă soluție este:

```
% gcc prob3.c -o prob3 -Wall -g
% gdb prob3
(gdb) break 113
(gdb) run
(gdb) print a
$1: { 7, 9, 3, 1, 0 <repeats 996 times> }
(gdb) quit
```

Pentru ca gdb să fie cât mai util trebuie să compilăm programul ca în prima linie din exemplu. După terminarea depanării, pentru obținerea executabilului final este recomandabil să compilăm programul cu opțiunile prezentate la începutul articolului.

Comanda break stabilește un punct de întrerupere la o anumită linie (în exemplu, 113) sau la prima linie a unei funcții (de exemplu: break main). Observăm de asemenea că print se descurcă foarte bine cu vectori, deci îl putem folosi pentru a afișa vectori oricât de mari, cu condiția ca numai un număr rezonabil de elemente să fie folosite.

Alte două comenzi utile sunt up și down. Acestea se deplasează în sus sau în jos pe stivă. Astfel, putem examina variabilele locale funcției care a apelat funcția curentă, sau funcției care a apelat-o la rândul ei pe aceasta etc.

Comenzile folosite pentru urmărirea cursului execuției programului sunt:

- continue - continuă execuția până la următorul punct de întrerupere;
- next - execută linia curentă;
- step - execută linia curentă, dar dacă acesta conține apelul unei funcții, "pășește" în interiorul funcției, oprindu-se la prima linie a acesteia;
- until - continuă execuția până când programul ajunge la o linie aflată după cea curentă. Este utilă spre exemplu pentru a trece peste o buclă neinteresantă. Poate primi și un argument, care este identic cu cel pentru break.

Comenzile anterioare sunt cele mai folosite, motiv pentru care este suficient să tastăm prima literă a comenzii în locul numelui complet.

Pentru a monitoriza constant evoluția unei valori, putem folosi comanda display. Aceasta este asemănătoare cu print, dar ea reevaluează și afișează o expresie după fiecare dintre aceste comenzi care execută secvențe din program. Mai trebuie menționat că gdb are o facilitate de *TAB-completion* foarte utilă, la fel ca și *bash*, iar informații suplimentare despre orice comandă se pot obține cu comanda help. Două comenzi utile nemenționate până acum sunt delete și undisplay, care anulează o comandă break sau display.

Rafinări ale limbajului C

Probabil cea mai evidentă schimbare față de mediile *Borland C* o reprezintă trecerea la întregi pe 32 de biți. A apărut și un tip nou, **long long**, care este un întreg pe 64 de biți. Compilatorul suportă și vectori locali de dimensiuni variabile:

```
int solve(int n){
  int a[n];
  ...
}
```

Această facilitate permite o scriere mai elegantă a codului și o economisire a memoriei (nu declarăm vectori având dimensiunile maxime posibile, ci vectori cu dimensiunile necesare). În timp ce folosirea acestei facilități nu are efecte negative pentru vectori (am întâlnit chiar programe care rulau semnificativ mai rapid când vectorii erau declarați astfel), ea duce la o oarecare încetinire a codului pentru matrice multidimensionale, deși de obicei diferența nu este semnificativă.

Programatorul trebuie să acorde atenție unui anumit aspect. Deși nu mai există restricțiile extrem de severe privitoare la memorie de sub *DOS*, comisia de evaluare stabilește unele restricții pentru memoria ce poate fi folosită de un program. Tendința este să se limiteze stiva la 1 MB de memorie, iar datele la 15 MB. Astfel, va trebui să avem grijă să nu încercăm să alocăm mai mult de 1 MB de spațiu pe stivă. Dacă este necesară mai multă memorie, tablourile de mari dimensiuni vor trebui declarate globale, fiind astfel mutate în zona de date, unde putem aloca 15 MB.

O facilitate importantă este atributul **inline** al funcțiilor. O funcție a cărei declarații este precedată de **inline** este integrată în corpul funcției apelante. Pentru funcții mici apelate des, aceasta poate duce la o sporire considerabilă a vitezei. Se poate împărți astfel programul în funcții fără restricții; se câștigă claritate și nu se pierde din viteză nici în cazul funcțiilor apelate des în bucle interioare.

Această facilitate trebuie folosită cu discernământ. Dacă încercăm să integrăm o funcție destul de complexă (cu multe variabile) într-o altă funcție complexă, codul va deveni mai lent. Acest lucru este datorat arhitecturii; procesoarele din generația *x86* au foarte puțini regiștri generali.

Vom discuta în continuare un aspect general valabil în orice mediu *C*, dar mai puțin cunoscute. În primul caz, problema este cauzată de faptul că limbajul *C* nu oferă decât vectori indexați începând cu zero. Mulți consideră acest fapt un dezavantaj, dar vom arăta că este doar un detaliu minor de sintaxă. Să presupunem că vrem să declarăm un vector care să fie indexabil cu valori între -100 și +150. Putem face asta foarte ușor astfel:

```
int a[251];
#define a               (a + 100)
```

Generalizând pentru indicii inferiori și superiori *X* și *Y*, va trebui să declarăm vectorul de mărimea (*Y*-*X*+1), lucru evident deoarece acesta este numărul de elemente al vectorului. Folosind o macrodefiniție ca în exemplu, preprocesorul va substitui orice referință la a prin (a+100). Acesta este un pointer la elementul care dorim să-l adresăm ca elementul 0. Elementul inferior al vectorului este (a+100)[-100], deci fizic a[0], iar elementul superior (a+100)[150], adică a[250], care este chiar ultimul element al vectorului. Acest truc ne poate ușura munca foarte mult în unele cazuri.

Mulți consideră că *C+Unix* reprezintă mediul ideal pentru programare. Astfel trecerea olimpiadelor internaționale pe aceste sisteme este una perfect justificată. Recomandăm și elevilor să nu ezite în a renunța la mediile de dezvoltare care rulează sub *DOS* și să se bucure de multiplele avantaje oferite de mediul *GNU/Linux*.

Mihai Pătrașcu este elev în clasa a XII-a la Liceul Carol I din Craiova. poate fi contactat prin e-mail la adresa mihai_p@pcnet.ro.