Lucrarea de Laborator Nr. 8

Programare in Linux

Curs: Utilizarea Sistemelor de Operare

Autor: Bardac Mircea Ionut

Versiune: 1.0.1/2005-11-23

Cuprins

1. Notiuni de baza....3
1.1. IDE = Integrated Development Environment....3
1.2. Debugger – gdb – The GNU Debugger....3
1.3. Debug symbols....3
1.4. Version Control....4
1.5. ANSI C....4
1.6. Core file....4
2. Debugging in Linux....4
2.1. Notiuni generale....4
2.2. Debugging „pasiv” (folosind core files)....6
2.3. Debugging „activ”....9
3. Detectia memory leak-urilor....11
3.1. Notiuni introductive....11
3.2. Valgrind....11
4. Utilizarea file descriptor-ilor....14
4.1. Terminologie....14
4.2. File descriptor-i standard....14
4.3. De ce nu facem piping mereu?....15
4.4. De ce sa folosim toti file descriptor-ii standard?....15
4.5. Exemplu de utilizare a file descriptor-ilor standard intr-un program....15
4.6. Exemple de utilizare a programului....16
5. Pipe-uri cu nume....17
5.1. Notiuni introductive....17
5.2. „Viata” pipe-urilor cu nume....17
5.3. Utilizarea pipe-urilor cu nume....18

Lista de modificari:

2005-11-21 v1.0 first release

2005-11-23 v1.0.1 spelling mistakes fixes, small rephrasings

Pachete Debian ce trebuie instalate: **gcc libc6-dev gdb valgrind**

Acest laborator are ca scop familiarizarea studentilor cu unelte de programare si tehnici de debugging/code-proofing in sistemul de operare Linux.

Se presupune ca studentii au suficiente cunostinte de programare de la cursurile paralele de specialitate astfel incat sa poata repara un program utilizand uneltele puse la dispozitie in Linux.

1. Notiuni de baza

1.1. IDE = Integrated Development Environment

Mediile de dezvoltare integrate (IDE) sunt aplicatii complexe utilizate de catre programatori, aplicatii care ofera acces la unelte precum:

- editor de cod (cu *syntax highlight*, *code folding*, *code completion* etc.)
- *contextual help* – ofera help adecvat contextului (informatii despre functia selectata in prezent etc.)
- acces facil la un compilator
- *debugger* – utilizat pentru detectarea bug-urilor din programe (rulare pas cu pas, *watches*, *call stack* etc.)
- *project manager* – utilizat pentru managementul proiectului (adaugat surse, configurat parametrii de compilare, editat automat fisiere Makefile etc.)
- *version control* (*code revisions*) – functie utilizata pentru lucrul cu mai multe versiuni, pe mai multe ramuri de dezvoltare etc.

Exemple de IDE-uri:

- Windows: Microsoft Visual Studio, DevC++ (http://www.bloodshed.net/)
- Linux: Kdevelop, Anjuta

More info: http://en.wikipedia.org/wiki/Integrated_development_environment

1.2. Debugger – gdb – The GNU Debugger

Debugger-ul consacrat in Linux este gdb. Este o aplicatie foarte puternica, cu foarte multe functii. Deoarece lucrul cu acesta nu este tocmai „placut”, au aparut numeroase front-end-uri pentru GDB, GDB-ul fiind integrat chiar si in IDE-uri. Aceste front-end-uri au profitat de intercomunicarea inter-procese si au „imbracat” GDB-ul intr-o interfata placuta utilizatorului.

More info: http://en.wikipedia.org/wiki/Debugger

1.3. Debug symbols

Simbolurile pentru debug (debug symbols) sunt informatii ce sunt incluse la compilare in executabilul rezultat, informatii folosite in procesul de debugging. Aceste informatii se refera in mod special la codul sursa, la simbolurile utilizate din biblioteci etc. Aceste simboluri sunt folosite de catre debugger pentru a identifica linia curenta in program, numele variabilelor etc.

1.4. Version Control

Controlul versiunilor reprezinta managementul mai multor versiuni pentru o unitate de informatie (orice fel de informatie, nu neaparat cod). Versiunile se identifica de obicei prin *revision number*, *revision level*. Pot exista *branch*-uri ale unui proiect, *tag*-uri cu versiuni etc.

Sistemele de *Revision Control* reprezinta in prezent suportul pentru dezvoltarea majoritatii proiectelor software.

Exemple de software pentru managementul versiunilor:widely used: **CVS**, **SVN** (Subversion); **Git** (folosit in dezvoltarea kernel-ului de Linux); **Darcs** etc.

More info:

- http://en.wikipedia.org/wiki/Revision_control
- http://en.wikipedia.org/wiki/List_of_revision_control_software

1.5. ANSI C

ANSI C este o versiune standard a limbajului de programare C. In programarea C este recomandata utilizarea functiilor ANSI C deoarece acestea asigura o compatibilitate crescuta intre diferite compilatoarele de C si implicit o portabilitate crescuta.

More info: http://en.wikipedia.org/wiki/ANSI_C

1.6. Core file

Un fisier core (*core file* sau *core dump*) este un fisier care contine imaginea memoriei unui proces la un moment dat, de obicei imediat dupa un crash.

More info: http://en.wikipedia.org/wiki/Core_dump

2. Debugging in Linux

2.1. Notiuni generale

Moduri de debugging

- „activ” - pornind aplicatia in debugger - `$ gdb comanda`
- „pasiv” - utilizand *core files* - `$ gdb --core=fisier_core comanda`

Cele doua moduri de lucru difera in principal prin abordarea aleasa pentru debugging:

- Primul mod se utilizeaza pentru debugging-ul pas cu pas, cu urmarirea modificarilor unor variabile etc.
- Modul al doilea este util atunci cand bug-urile sunt extrem de rare si se soldeaza cu oprirea aplicatiei. Presupunand ca o aplicatie isi inceteaza activitatea din cauza unui acces neprotejat la memorie (in acel moment SO-ul opreste executia aplicatiei), un fisier *core* este creat in care se salveaza starea starea aplicatiei la momentul opririi. Prin deschiderea acestui *core file* in **gdb**, se poate face o analiza „statica” a unui bug (urmarind *call stack*-ul, valoarile variabilelor etc.).

Debugging-ul este mult ingreunat de lipsa simbolurilor pentru debug. Acestea se activeaza la compilarea cu GCC-ul folosind parametrul **-g**. In mod implicit, dimensiunea executabilelor va creste deoarece simbolurile pentru debug vor fi incluse in acestea.

Vom lua pentru exemplificare urmatorul fisier:

testfile.c

```
#include <stdio.h>

int main()
{
        FILE *f=fopen("fisier_nou.txt","r");
        //if (!f) return 1;
        fclose(f);
        return 0;
}
```

Il vom compila fara cu debug symbols o data (este posibil ca GCC-ul sa considere implicita optiunea -g in anumite cazuri):

```
$ gcc testfile.c -g -o testfile1
$ file testfile1
testfile1: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
GNU/Linux 2.4.1, dynamically linked (uses shared libs), for GNU/Linux 2.4.1,
not stripped
```

Observati ca **file** indica faptul ca fisierul executabil include simboluri pentru debug utilizand sintagma „not stripped”.

Pentru a elimina simbolurile pentru debug, se poate folosi comanda **strip**.

```
$ gcc testfile.c -o -g testfile2
$ strip testfile2
$ file testfile2
testfile2: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
GNU/Linux 2.4.1, dynamically linked (uses shared libs), for GNU/Linux 2.4.1,
stripped
$ ls -l testfile* | grep x
-rwxr-xr-x  1 root root 6957 2005-11-21 21:32 testfile1
-rwxr-xr-x  1 root root 3096 2005-11-21 21:34 testfile2
```

Se observa:

- output-ul modificat al comenzii **file** (fisierul *testfile2* nu are simboluri pentru debug – este „stripped”)
- dimensiunea fisierului fara simboluri de debug este mai mica

Nota:

- pentru simplitatea comenzii, in compilarea de mai sus nu s-a mai utilizat parametrul **-Wall** – se recomanda totusi ca, de fiecare data, cel putin in procesul de dezvoltare, compilarile sa se faca activand toate warning-urile cu acest parametru

2.2. Debugging „pasiv” (folosind *core files*)

Pentru inceput, vom presupune ca vrem sa studiem un bug foarte rar, intr-o aplicatie cu debug symbols incluse si nu facem decat sa pornim aplicatia. Aplicatia urmeaza sa se „crash-uiasca” la un moment dat dar, avand *core files* active, vom putea reface lista ultimelor apeluri efectuate si locul unde s-a produs bug-ul.

Pentru a activa *core files*-urile, vom utiliza urmatoarea comanda:

```
# ulimit -c unlimited
```

ulimit este un program pentru configurarea limitelor unor anumiti parmetrii la nivelul kernel-ului. Modificarea acestor parametrii nu poate fi intotdeauna efectuata de catre utilizatori non-root. Printre acesti parametrii se gaseste si dimensiunea unui *core file*. **-c unlimited** specifica faptul ca vrem ca dimensiunea unui core file sa fie nelimitata. Pentru mai multe informatii despre limite, folositi comanda **man ulimit**.

Configurarea folosind comanda **ulimit** nu este permanenta. Pentru a face permanenta aceasta configurare se poate modifica

- fisierul */etc/profile* prin adaugarea comenzii la sfarsitul acestuia
- fisierul */etc/security/limits.conf* prin adaugarea unei linii corect formatate, conforma cu specificatiile date in fisier

In continuarea vom analiza modificand temporar dimensiunea maxima a unui **core file**. Vom realiza urmatorul test ca **root** (veti observa „#” in fata comenzilor) pentru a fi sigur ca aceasta comanda va fi permisa. Vom dezactiva intai *core file*-ul pentru a se putea vedea diferenta pe care o introduce aceasta configurare la rulare (mai precis la *crash*-ul a unei aplicatii).

```
# ulimit -c 0
# ./testfile1
Segmentation fault
# ulimit -c unlimited
# ./testfile1
Segmentation fault (core dumped)
```

„(core dumped)” semnifica faptul ca s-a realizat o imagine a memoriei procesului in momentul in care s-a produs *segmentation fault*-ul. Aceasta imagine se gaseste implicit in fisierul cu numele „core” din directorul de unde s-a pornit aplicatia.

Pentru a analiza fisierul *core*, vom proni GDB-ul cu urmatoarea comanda:

```
# gdb -core=core ./testfile1
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...Using host libthread_db
library "/lib/tls/libthread_db.so.1".

Core was generated by `./testfile1'.
Program terminated with signal 11, Segmentation fault.

warning: current_sos: Can't read pathname for load map: Input/output error

Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0xb7e1f047 in fclose@@GLIBC_2.1 () from /lib/tls/libc.so.6
(gdb) bt full
#0  0xb7e1f047 in fclose@@GLIBC_2.1 () from /lib/tls/libc.so.6
No symbol table info available.
#1  0x08048412 in main () at testfile.c:7
        f = (FILE *) 0x0
```

Pentru a iesi din GDB, se utilizeaza combinatia de taste **Ctrl+D**.

Se observa faptul ca gdb-ul, dupa pornire, afiseaza un prompt la care se pot introduce comenzi. Pentru a vedea setul de comenzi GDB, puteti introduce comanda **help**. Comenzile sunt impartite pe categorii.

Spre exemplu, pentru a afla informatii despre comenzile pentru studiul datelor, se va folosi comanda **help data**. Veti descoperi astfel ca exista o serie de comenzi pentru inspectia/modificarea datelor/codului. Comenzi mai uzuale sunt: **print**, **display** si **set** (scurt help gasiti utilizand **help nume_comanda**).

Observati de asemenea in output-ul de mai sus utilizarea comenzii **bt full**. **bt** este prescurtarea lui **backtrace**. GDB-ul accepta si prescurtari pentru comenzi. Backtrace intoarce lista apelurilor efectuate pana in momentul opririi (*call stack*-ul). In varful stivei, in *frame*-ul #0, gasiti apelul functiei fclose() din biblioteca de functii C. La baza stivei, in *frame*-ul #1, gasiti functia main() si locatia de unde s-a facut apelul catre functia fclose().

Frame-ul (sau *stack frame*-ul) este un element din call stack (stiva apelurilor), adica o functie. Debugging-ul se poate face in frame-uri diferite deoarece, spre exemplu, fiecare frame are variabilele lui.

Se mai observa folosirea parametrului **full** impreuna cu comanda **bt**. Acesta induce afisarea tuturor variabilelor din fiecare frame si valorile acestora.

Sa vedem ce s-ar fi intamplat daca am fi avut fisierele core active dar am fi folosit un executabil fara debug symbols (in cazul nostru *testfile2*).

```
# ./testfile2
Segmentation fault (core dumped)

# gdb -core=core ./testfile2
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...(no debugging symbols found)
Using host libthread_db library "/lib/tls/libthread_db.so.1".

Core was generated by `./testfile2'.
Program terminated with signal 11, Segmentation fault.

warning: current_sos: Can't read pathname for load map: Input/output error

Reading symbols from /lib/tls/libc.so.6...(no debugging symbols found)...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0xb7eb2047 in fclose@@GLIBC_2.1 ()
   from /lib/tls/libc.so.6
(gdb) bt full
#0  0xb7eb2047 in fclose@@GLIBC_2.1 () from /lib/tls/libc.so.6
No symbol table info available.
#1  0x08048412 in ?? ()
No symbol table info available.
#2  0x00000000 in ?? ()
No symbol table info available.
#3  0x08048508 in _IO_stdin_used ()
No symbol table info available.
#4  0xb7e88085 in __new_exitfn () from /lib/tls/libc.so.6
No symbol table info available.
#5  0xb7e71d6b in __libc_start_main () from /lib/tls/libc.so.6
No symbol table info available.
#6  0x08048351 in ?? ()
No symbol table info available.
(gdb)

# file /lib/tls/libc.so.6
/lib/tls/libc.so.6: symbolic link to `libc-2.3.5.so'

# file /lib/libc-2.3.5.so
/lib/libc-2.3.5.so: ELF 32-bit LSB shared object, Intel 80386, version 1
(SYSV), for GNU/Linux 2.4.1, not stripped
```

Observam ca debugger-ul este oarecum „confuz”, el nestiind simbolurile existente in executabil si nici numele unor functii. Avem totusi noroc ca biblioteca standard C include simboluri pentru debugging. Majoritatea bibliotecilor sunt compilate astfel pentru a usura debugging-ul.

2.3. Debugging „activ”

Pentru o analiza pas cu pas, vom utiliza executabilul cu simboluri pentru debug.

```
$ gdb ./testfile1
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...Using host libthread_db
library "/lib/tls/libthread_db.so.1".

(gdb) break main
Breakpoint 1 at 0x80483f0: file testfile.c, line 5.
```

Am utilizat comanda **break** din GDB pentru a activa un breakpoint. La rulare, programul se va opri la intalnirea unui breakpoint. In cazul nostru, am pus un breakpoint pe functia main. Breakpoint-urile se pot specifica si prin numarul line unde se doreste oprirea si pot fi (sau nu) conditionate (ex: „break 5 a > 6” = break la linia 5 daca a>6). Pentru mai multe informatii utilizati comanda gdb **help break**.

Am folosit comanda **run** pentru a porni programul. Dupa cum ne asteptam, programul s-a intrerupt la primul breakpoint:

```
(gdb) run
Starting program: /home/mircea/pub/uso/lab-testing/lab8/testfile1

Breakpoint 1, main () at testfile.c:5
5               FILE *f=fopen("fisier_nou.txt","r");
```

In continuare, am utilizat comanda **next** de doua ori pana cand a aparut bug-ul. Ca alternativa la next, se poate utiliza **step**. Exista o mica diferenta intre ele: comanda **step** intra in subrutine (face *step in*), pe cand **next** trece peste acestea (face *step over*).

```
(gdb) next
7               fclose(f);
(gdb) next

Program received signal SIGSEGV, Segmentation fault.
0xb7eaf047 in fclose@@GLIBC_2.1 () from /lib/tls/libc.so.6
```

Programul s-a oprit in acelasi loc ca si mai inainte. De aceasta data vom folosi o alta abordare. Vom analiza manual valorile variabilelor din contextul programului nostru.

```
(gdb) print f
No symbol "f" in current context.
(gdb) frame
#0  0xb7eaf047 in fclose@@GLIBC_2.1 () from /lib/tls/libc.so.6
(gdb) frame 1
#1  0x08048412 in main () at testfile.c:7
7               fclose(f);
(gdb) print f
$1 = (FILE *) 0x0
(gdb)
```

Intai trebuie sa modificam frame-ul curent. Variabila *f* nu se gaseste in frame-ul #0. In frame-ul #1 (=functia *main()*), utilizand comanda **print** a GDB-ului descoperim ca variabila *f* este nula.

Dupa cum se stie si din laboratorul anterior, corectia la aceasta problema se realizeaza prin testarea variabilei *f* inainde de apelul functiei fclose.

Programul utilizat anterior devine in acest fel:

testfile.c

```
#include <stdio.h>

int main()
{
        FILE *f=fopen("fisier_nou.txt","r");
        if (!f) return 1;
        fclose(f);
        return 0;
}
```

Un tutorial extins de GDB:

- http://www.dirac.org/linux/gdb/

Exercitiu de GDB (bun de asemenea ca tutorial):

- http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html#tth_sEc4

3. Detectia memory leak-urilor

3.1. Notiuni introductive

Uneori, apare nefericita situatie cand un program de-al nostru incepe sa ocupe memorie fara un motiv evident. Efectul este de obicei sesizat de abia dupa ce toata memoria computerului se umple si sistemul devine instabil.

Situatia apare in momentul in care un programator uita sa elibereze memoria alocata dinamic pentru variabile din program. Uneori, contextul de lucru poate fi destul de complicat si numarul de variabile alocate dinamic poate ingreuna detectia acestor leak-uri.

3.2. Valgrind

Pentru a preveni astfel de accidente, se utilizeaza programe specializate de analiza. Un astfel de program este si **valgrind**.

Valgrind este o masina virtuala in care se ruleaza programul ce urmeaza a fi analizat. In masina virtuala, programul este supus unor teste. Masina virtuala converteste codul masina al programului executat intr-un format intermediar. Ulterior, acest format este trecut printr-un tool ales de catre utilizator. In functie de complexitatea tool-ului, rularea programului poate fi incetinita foarte mult deoarece o mare parte din instructiuni trebuie interpretate.

„Sintaxa” pentru rularea unui program sub valgrind este urmatoarea:

```
$ valgrind –tool=nume_tool comanda_aplicatie_cu_parametrii
```

Tool-uri disponibile in valgrind (le-am afisat folosind o sintaxa gresita pentru apelarea comenzii **valgrind**):

```
$ valgrind --tool=
Can't open tool "": /usr/lib/valgrind/vgtool_.so: cannot open shared object
file: No such file or directory
valgrind: couldn't load tool
Available tools:
        memcheck
        addrcheck
        cachegrind
        none
        callgrind
        helgrind
        lackey
        massif
```

Vom folosi un mic program de test:

leak.c

```
#include <stdio.h>
#include <malloc.h>

int main(){
        int *p;
        int i;
        for (i=0;i<10;i++){
                p=(int *)malloc(sizeof(int));
                *p=i*i;
                printf("%d ",*p);
        }
        printf("\n");
        return 0;
}
```

Il vom compila cu debug symbols, pentru a putea primi informatii relevante de la **valgrind**.

```
$ gcc leak.c -g -o leak
$ ./leak
0 1 4 9 16 25 36 49 64 81
```

Am si rulat programuil si am vazut ca functioneaza. Nimic suspect.

Exemplul de mai sus este foarte „inocent”. In conditii reale, unde codul sursa este mult mai mare si programul se executa mai mult timp, o astfel de „scurgere” poate fi greu detectabila si poate avea efecte nedorite.

Vom rula programul „leak” sub valgrind folosind comanda

```
$ valgrind --tool=memcheck --leak-check=full ./leak
==28464== Memcheck, a memory error detector.
==28464== Copyright (C) 2002-2005, and GNU GPL'd, by Julian Seward et al.
==28464== Using LibVEX rev 1367, a library for dynamic binary translation.
==28464== Copyright (C) 2004-2005, and GNU GPL'd, by OpenWorks LLP.
==28464== Using valgrind-3.0.1, a dynamic binary instrumentation framework.
==28464== Copyright (C) 2000-2005, and GNU GPL'd, by Julian Seward et al.
==28464== For more details, rerun with: -v
==28464==
0 1 4 9 16 25 36 49 64 81
==28464==
==28464== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 20 from 4)
==28464== malloc/free: in use at exit: 40 bytes in 10 blocks.
==28464== malloc/free: 10 allocs, 0 frees, 40 bytes allocated.
==28464== For counts of detected errors, rerun with: -v
==28464== searching for pointers to 10 not-freed blocks.
==28464== checked 69632 bytes.
==28464==
==28464== 40 bytes in 10 blocks are definitely lost in loss record 1 of 1
==28464==    at 0x1B9048A2: malloc (in /usr/lib/valgrind/vgpreload_memcheck.so)
==28464==    by 0x80483E4: main (leak.c:8)
==28464==
==28464== LEAK SUMMARY:
==28464==    definitely lost: 40 bytes in 10 blocks.
==28464==      possibly lost: 0 bytes in 0 blocks.
==28464==    still reachable: 0 bytes in 0 blocks.
==28464==         suppressed: 0 bytes in 0 blocks.
==28464== Reachable blocks (those to which a pointer was found) are not shown.
==28464== To see them, rerun with: --show-reachable=yes
```

(author's highlight)

Observam ca valgrind, utilizand tool-ul **`memcheck`**, a detectat unde se face o alocare de memorie, memorie care ulterior nu este eliberata. Pentru tool-ul **`memcheck`** s-a utilizat si parametrul **`--leak-check=full`** pentru a induce afisarea pe ecran a locatiei unde se gaseste *memory leak*-ul.

Memory leak-ul se poare repara prin aplicarea urmatorului patch

leakfix.patch

```
--- leak.c      2005-11-22 00:11:49.000000000 +0200
+++ leak_fix.c  2005-11-22 00:12:35.000000000 +0200
@@ -8,6 +8,7 @@
                p=(int *)malloc(sizeof(int));
                *p=i*i;
                printf("%d ",*p);
+               free(p);
        }
        printf("\n");
        return 0;
```

folosind una dintre urmatoarele comenzi (comenzi echivalente in acest context)

```
$ patch < leakfix.path
$ patch leak.c leakfix.patch
```

Vom compila si executa din nou programul, atat individual cat si in valgrind

```
$ gcc leak.c -g -o leak
$ ./leak
0 1 4 9 16 25 36 49 64 81
$ valgrind --tool=memcheck --leak-check=full ./leak
==29068== Memcheck, a memory error detector.
==29068== Copyright (C) 2002-2005, and GNU GPL'd, by Julian Seward et al.
==29068== Using LibVEX rev 1367, a library for dynamic binary translation.
==29068== Copyright (C) 2004-2005, and GNU GPL'd, by OpenWorks LLP.
==29068== Using valgrind-3.0.1, a dynamic binary instrumentation framework.
==29068== Copyright (C) 2000-2005, and GNU GPL'd, by Julian Seward et al.
==29068== For more details, rerun with: -v
==29068==
0 1 4 9 16 25 36 49 64 81
==29068==
==29068== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 20 from 4)
==29068== malloc/free: in use at exit: 0 bytes in 0 blocks.
==29068== malloc/free: 10 allocs, 10 frees, 40 bytes allocated.
==29068== For counts of detected errors, rerun with: -v
==29068== No malloc'd blocks -- no leaks are possible.
```

4. Utilizarea file descriptor-ilor

4.1. Terminologie

Termenul de „*file descriptor*” se intalneste mai ales in sistemele de operare POSIX. In Microsoft Windows si in biblioteca stdio (C) se prefera termenul de „file *handler*”.

More info: http://en.wikipedia.org/wiki/File_descriptor

4.2. File descriptor-i standard

Exista 3 *file descriptor*-i standard:

- 0 = stdin (intrarea standard, de obicei tastatura)
- 1 = stdout (iesirea standard, de obicei ecranul)
- 2 = stderr (iesirea pentru erori standard, de obicei ecranul)

Spunem „de obicei” pentru ca toti acesti descriptori pot fi redirectionati:

- unii catre altii (exemplu: stderr catre stdiout)
- catre fisiere (exemplu: stderr intr-un fisier, stdout in alt fisier)
- catre alte comenzi, prin utilizarea pipe-urilor

4.3. De ce nu facem piping mereu?

Piping-ul se prezinta sub urmatoarea forma

```
$ comanda1 | comanda2
```

(comenzile pot avea evident si parametrii)

Piping-ul se realizeaza intre 2 procese. Modul de functionare este foarte simplu: **stdout**-ul *comenzii1* este condus („pipe-uit”) in **stdin**-ul *comezii2*.

Se pot face si lanturi de comenzi utilizand mai multe pipe-uri. Intregul lant are un **stdin** (stdin-ul primei comenzi) si un **stdout** (stdout-ul ultimei comenzi).

File descriptorii se utilizeaza pentru a conecta procesele cu datele. Datele sunt de obicei prezente sub forma de fisiere sau pseudo-fisiere.

4.4. De ce sa folosim toti file descriptor-ii standard?

Sistemul de operare Linux, *by design*, a fost gandit sa fie modular. Sistemul de operare „traieste” din interactiunea mai multor componente, o parte dintre acestea fiind procese. Procesele fac uz permanent de mijloacele de comunicare inter-procese.

In momentul in care dorim ca o aplicatie sa se integreze bine intr-un sistem, trebuie sa pornim de la ce informatii trebuie sa schimbe aceasta cu restul sistemului si cum anume poate sa faca acest lucru. Aceasta „interactiune” este definita prin rolul file descriptor-ilor standard.

4.5. Exemplu de utilizare a *file descriptor*-ilor standard intr-un program

Avem in continuare un exemplu de program care face uz de toti file descriptorii standard:

fds.c

```
#include <stdio.h>

int main(int argc, char* argv[]){
        int c=0;
        char s[1024];
        while (!feof(stdin)){
                fgets(s,1024,stdin);
                c++;
        }
        fprintf(stdout,"%d\n",c); // afisam numarul de linii
        fprintf(stderr,"Numarul de linii introduse la intrare este: %d\n",c);
         // afisam "friendly" numarul de linii
        return 0;
}
```

Ce observam:

- exista o singura intrare – **stdin** – de unde se citesc date
- exista o iesire normala – **stdout**- unde se afiseaza DOAR rezultatul (date de iesire)
- la iesirea pentru erori – **stderr –** este afisat un mesaj, nu doar datele de iesire

De ce anume am utiliza astfel descriptorii?
Sa compilam programul mai intai si sa ne „producem” niste date de intrare:

```
$ gcc fds.c -o fds
$ cp fds.c fds.in
```

4.6. Exemple de utilizare a programului

```
$ cat fds.in | ./fds
15
Numarul de linii introduse la intrare este: 15
```

comanda echivalenta cu

```
$ ./fds <fds.in
15
Numarul de linii introduse la intrare este: 15
```

Continutul fisierului *fds.in* este trecut in prin **stdin**-un programului nostru. Pe ecran este afisat atat ce am scris pe **stdout** cat si ce am scris pe **stderr**.

Vom renunta in continuare la piping, pentru a fi mai evidenta redirectionarea *file descriptor*-ilor standard.

```
$ ./fds <fds.in >fds.out
Numarul de linii introduse la intrare este: 15
$ cat fds.out
15
```

Acesta este modul standard de executie a unui program supus testarii. Programul are ca intrare fisierul „fds.in” si ca iesire fisierul „fds.out”. Pe ecran raman afisate erorile. Iesirea poate fi comparata astfel (poate folosind **`diff`**) cu o iesire corecta.

Se poate face redirectionarea erorilor catre un alt fisier astfel:

```
$ ./fds <fds.in >fds.out 2>fds.err
$ cat fds.out
15
$ cat fds.err
Numarul de linii introduse la intrare este: 15
```

Pe ecran nu va mai fi afisat nimic. Observati „2” inainte de „>” - astfel se indica numarul *file descriptor*-ului ce va fi redirectionat.

Se poate face redirectionarea atat a lui **stdout** cat si a lui **stderr** in acelasi fisier :

```
$ ./fds <fds.in >fds.all 2>&1
$ cat fds.all
15
Numarul de linii introduse la intrare este: 15
```

Pe ecran nu va fi afisat nimic. **stderr** este redirectionat catre **stdout** folosind „2>&1”.
Atentie! Ordinea parametrilor conteaza: Daca „2>&1” nu este la sfarsit, nu se va stii ca *stderr* are aceeasi destinatie (fisier) ca si *stdout*.

Erorile pot fi ignorate, prin redirectionarea lor catre un pseudo-fisier

```
$ ./fds <fds.in >fds.out 2>/dev/null
```

„/dev/null” poate fi privit ca un sac fara fund, o gaura neagra, unde se poate redirectiona orice si unde totul dispare imediat ce a intrat.

Alte pseudo-fisiere „uzuale” mai sunt:

- /dev/zero – fisier ce poate fi folosit la intrare; se citeste numai zero; nu are capat
- /dev/random, /dev/urandom – fisiere ce pot fi folosite la intrare; informatiile ce se citesc din ele sunt produse de catre sistemul de operare si sunt aleatoare (Atentie! generarea de date aleatoare este consumatoare de timp de procesor)

5. Pipe-uri cu nume

5.1. Notiuni introductive

Pipe-urile cu nume (numite si *FIFOs* datorita comportarii) sunt extensii ale pipe-urilor traditionale si reprezinta si ele mijloace de comunicare inter-procese. Pipe-urile normale se mai numesc si anonime, deoarece exista doar pe perioada vietii celor doua procese se se gasesc la capete.

More info: http://en.wikipedia.org/wiki/Named_pipe

5.2. „Viata” pipe-urilor cu nume

Pentru a crea un pipe cu nume se utilizeaza comanda **mkfifo.**

```
$ mkfifo mypipe
```

Se poate vedea utilizand „ls -l” ca in directorul curent apare un fisier special, cu tipul „p”

```
$ ls -l mypipe
prw-r--r--  1 mircea users 0 2005-11-22 01:28 mypipe
```

Din moment ce pipe-urile cu nume se prezinta sub forma de fisiere, in momentul in care nu se mai doreste utilizarea lor acestea se pot sterge folosind comanda **rm**. Ele nu dispar din sistemul de fisiere in momentul in care nu mai au nimic la capete.

5.3. Utilizarea pipe-urilor cu nume

Vom face un mic experiment, utilizand 2 console/terminale si pipe-ul cu nume creat anterior. Vom rula urmatoarele comenzi, in ordinea de mai jos:

- consola 2

```
$ ./fds <mypipe 2>/dev/null
```

- consola 1

```
$ cat fds.in >mypipe
```

Ce vor face aceste doua comenzi?

Vom interpreta rezultatele in ordinea inversa a comenzilor, deoarece prima comanda citeste din pipe si ultima scrie. Prima comanda este in asteptare, atat timp cat nimeni nu se scrie in pipe.

1. Ultima comanda scrie in pipe continutul fisierului *fds.in*.
2. Comanda din consola 2 porneste de fapt programul anterior nostru, cu **stdin** din **mypipe**, **stdout** pe ecran si **stderr** in */dev/null*; comanda mentionata anterior a pus ceva in **mypipe** si programul nostru a afisat pe ecran numarul de linii pe care le-a citit din mypipe (=numarul de linii din *fds.in*)

Citirea simultana a aceluiasi named pipe de catre mai multe procese este imposibila. Datele scrise in pipe ajung doar la unul dintre procese care citesc din pipe.