Operacions bàsiques d'entrada/sortida a disc i *buffer overflow*

Lluís Garrido – lluis.garrido@ub.edu

Setembre 2014

En aquesta segona entrega descriurem breument les funcions bàsiques per realitzar operacions d'entrada sortida al disc i explicarem què és el problema del *buffer overflow* donant alguns consells per evitar-lo.

Índex

1 Funcions bàsiques d'accés a disc

En aquesta secció ens centrarem en algunes funcions de la llibreria estàndard d'entrada/-sortida. Aquesta llibreria està especificada per l'estàndard ISO C i està implementada per molts sistemes operatius apart dels sistemes Unix. Al llenguatge C, per poder fer servir les funcions incloses en aquesta llibreria hem d'incloure aquesta instrucció

```
#include <sdtio.h>
```

Les funcions incloses a la llibreria ens permeten realitzar operacions d'entrada/sortida. La gran majoria de les vegades aquesta llibreria es fa servir per manipular fitxers de disc (obrir fitxers, guardar o llegir informació dels fitxers, etc.). La llibreria permet, però, tractar amb fluxos de dades que no necessàriament han d'estar associats a fitxers de disc. Els fluxos poden estar associats també, per exemple, a operacions de connexió remota amb un altre ordinador. També poden estar associats al teclat o la pantalla, veure secció 1.3.

1.1 Descriptors de fitxers

En un sistema operatiu Unix, tots els fitxers oberts per un procés (estiguin associats a un fitxer disc, la xarxa, el teclat, la pantalla, ...) tenen associat un descriptor de fitxer. Un descriptor de fitxer és simplement un sencer no negatiu. Quan obrim un fitxer existent o en creem un de nou, el sistema operatiu retorna al procés el descriptor de fitxer associat. Quan es vol escriure o llegir d'aquest fitxer, el sistema operatiu utilitza aquest identificador per saber on ha d'escriure o llegir les dades.

1.2 Obrir i tancar un flux de dades

Per obrir un flux de dades i, en particular, un fitxer, hem de fer servir la següent instrucció

```
FILE *fopen(const char *path, const char *type);
```

La funció *fopen* permet obrir un fitxer amb el nom *path* i amb el mode especificat a *type*. En cas que el fitxer no es pugui obrir, la funció retorna un punter *NULL*. Per a informació sobre els modes en què es pot obrir un fitxer (lectura, escriptura, etc.) consultar el manual amb

```
$ man fopen
```

Quan obrim un flux de dades, la funció estàndard d'entrada/sortida *fopen* retorna un punter a una estructura de tipus *FILE*. Aquesta estructura conté tota la informació requerida per a la llibreria estàndard per manipular el flux de dades: el descriptor de fitxer associat al flux d'entrada/sortida (veure secció 1.1), un punter a una memòria intermitja per al flux de dades, la mida de la memòria intermitja, el nombre de bytes emmagatzemat a la memòria intermitja, i altres dades.

El flux de dades es pot tancat mitjançant l'operació *fclose*

```
int fclose(FILE *fp);
```

Les aplicacions de programari no haurien de manipular mai directament l'estructura *FILE*. Aquesta estructura es passa com a argument a les funcions d'entrada/sortida que ens ofereix la llibreria, alguna de les quals veurem ara.

1.3 Entrada estàndard, sortida estàndard i sortida estàndard d'error

Els fluxos d'entrada estàndard, sortida estàndard i sortida estàndard d'error es defineixen de forma automàtica pel sistema operatiu a l'hora de crear un procés: entrada estàndard, sortida estàndard i sortida estàndard d'error. Normalment l'entrada estàndard està associada al teclat, mentre que la sortida estàndard i la sortida estàndard d'error està associada a un terminal de la pantalla. Aquests tres fluxos estan associats a les variables predefinides de tipus *FILE* anomenades *stdin*, *stdout* i *stderr*. Aquests tres fitxers s'obren automàticament pel sistema operatiu en crear el procés, i estan definits en tot el codi. Així, si en un programa escrivim dades en *stdout* aconseguirem que aquestes dades apareguin per pantalla. De la mateixa forma, si llegim dades de *stdin*, estarem llegint les dades del teclat. Més endavant veurem alguns exemples.

1.4 Funcions per llegir i escriure

La llibreria estàndard d'entrada/sortida ofereix a l'usuari múltiples funcions per a manipular fluxos de dades. Els més importants, per a la realització de les pràctiques, es llisten a continuació. Per a informació específica sobre cada funció, utilitzar la instrucció `man` des del terminal.

En aquesta secció farem la distinció entre dos tipus de flux: el *formatat* i el *no formatat*. El no formatat és la forma més bàsica d'entrada/sortida: en aquest cas es transfereix la representació binària interna de les dades directament entre memòria i el fitxer. Així, per exemple, si s'escriu un *double* a disc, s'escriuran 8 bytes, que són els bytes que ocupa a memòria. L'entrada/sortida no formatada és la forma més compacta per emmagatzemar dades.

La sortida formatada converteix la representació interna de les dades a caràcters ASCII que són escrits al fitxer de sortida. L'entrada/sortida formatada acostuma a ocupar més en disc però és directament llegible per un humà. Un fitxer formatat és aquell que es pot visualitzar en un terminal o editar en un fitxer de text pla. Així, en escriure un *double* a disc s'hi escriurà la seva representació en decimal (en comptes d'escriure 8 bytes).

La diferència entre tots dos no es troba en la forma d'obrir el fitxer, sinó en les funcions que s'utilitzen per escriure-hi o llegir dades.

Cal esmentar (important!) que en el cas d'una cadena de caràcters la representació formatada és igual a la representació no formatada. En el cas de llegir un fitxer amb cadenes de caràcters es recomana utilitzar les funcions no formatades.

1.4.1 Entrada/sortida no formatada

Hi ha tres tipus de funcions per a tractar fitxers no formatats

1. Operacions amb un sol caràcter (o byte). Podem llegir i escriure un caràcter o byte amb les funcions: *fgetc*, *getchar*, *fputc* i *putchar*. Observar al manual que s'hi indica que *getchar* és equivalent a fer *fgetc(stdin)*, i que *putchar(c)* és equivalent a fer *fputc(c, stdout)*.

2. Operacions amb una línia. Si volem llegir o escriure una línia a la vegada, podem utilitzar *fgets*, *gets*, *fputs* i *puts*. Cada línia s'acaba amb un retorn de carro, i a la funció *fgets* s'ha d'especificar el nombre màxim de caràcters que volem llegir.

3. Entrada/sortida directa. Aquests tipus d'operacions està suportat per les funcions *fread* i *frwrite*. Aquestes dues funcions permeten llegir o escriure un determinat

```c
#include <stdio.h>

#define MAXLINE 100

int main(void)
{
  char buf[MAXLINE];
  while (fgets(buf, MAXLINE, stdin) != NULL)
    fputs(buf, stdout);

  return 0;
}
```

Figura 1: Codi amb *fgets* (veure codi `exemple1.c`).

nombre de bytes del fitxer. Aquestes funcions són particularment útils per llegir o escriure tipus agregats (estructures de dades, vectors, etc.)

En aquesta secció ens centrarem només en el 2n i 3r punt.

Funció fgets A la figura 1 es mostra un codi que agafa una línia l'entrada estàndard *stdin* i la copia a la sortida estàndard *stdout*. Un cop executat el programa podem sortir-ne polsant "Control-D" (equivalent a "tancar un fitxer" per teclat; així *fgets* retorna NULL). Observeu que estem llegint dades de *stdin* i escrivint-les a *stdout*, i que no cal declarar en cap moment aquestes variables. Estan definides automàticament en el moment de crear el procés (veure secció 1.3).

Funcions fwrite i fread Mostrem a continuació un exemple amb les funcions *fwrite* i *fread*. En particular, aquí teniu un exemple on escrivim un vector de *double* a un fitxer, veure figura 2.

Mitjançant el primer *fwrite* estem escrivint al fitxer el nombre d'elements que conté el vector i mitjançant el segon *fwrite* estem escrivint tot el vector *a*. Recordeu que *fwrite* escriu el fitxer en format no formatat (és a dir, tal com les dades estan guardades a memòria). En particular, si intenteu editar amb editor de text qualsevol el fitxer vector.data no hi veureu més que bytes sense cap sentit. No és més que la representació a la CPU dels valors guardats.

Amb l'exemple de la figura 3 llegirem les dades escrites per l'exemple anterior. Observeu que la funció *fread* retorna el nombre d'elements que s'han pogut llegir correctament. D'aquesta forma sabem si el fitxer està ben format i si realment s'han llegit totes les dades previstes.

1.4.2 Entrada/sortida formatada

La sortida formatada es pot realitzar mitjançant les següents dues funcions: *printf* i *fprintf*. La primera funció permet escriure cap a la sortida estàndard (*stdout*) i la segona permet escriure cap al flux de dades especificat als arguments (típicament un fitxer a disc). Per a l'entrada formatada, disposem de les funcions *scanf* (que llegeix de l'entrada estàndard) i *fscanf* (que permet llegir dades formatades d'un fitxer o flux de dades).

Funcions fscanf i fprintf Vegeu la figura 4, hi veiem el codi que agafa una línia l'entrada estàndard i la copia a la sortida estàndard. Un cop executat el programa podem sortir-ne polsant "Control-D". Compareu aquest programa amb el de la secció 1.4.1 (que

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char **argv)
{
  FILE *fp;
  int i, size;
  double *a;

  size = 10;
  a = malloc(sizeof(double) * size);

  for(i = 0; i < size; i++)
    a[i] = sqrt((double) i);

  fp = fopen("vector.data", "w");

  if (!fp) {
    printf("ERROR: no he pogut obrir el fitxer.\n");
    exit(EXIT_FAILURE);
  }

  fwrite(&size, sizeof(int), 1, fp);
  fwrite(a, sizeof(double), size, fp);

  fclose(fp);

  return 0;
}
```

Figura 2: Codi amb fwrite (veure codi `exemple2.c`).

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  FILE *fp;
  int i, size, nelems;
  double *a;

  fp = fopen("vector.data", "r");

  if (!fp) {
    printf("ERROR: no he pogut obrir el fitxer.\n");
    exit(EXIT_FAILURE);
  }

  nelems = fread(&size, sizeof(int), 1, fp);

  if (nelems != 1) {
    printf("ERROR: el fitxer és massa curt.\n");
    exit(EXIT_FAILURE);
  }

  if (size <= 0) {
    printf("ERROR: la mida no és correcta.\n");
    exit(EXIT_FAILURE);
  }

  a = malloc(sizeof(double) * size);

  nelems = fread(a, sizeof(double), size, fp);

  if (nelems != size) {
    printf("ERROR: el fitxer és massa curt.\n");
    exit(EXIT_FAILURE);
  }

  fclose(fp);

  printf("Hi ha %d valors guardats al fitxer.\n", nelems);
  printf("Els valors son: ");

  for(i = 0; i < 10; i++)
    printf(" %f, ", a[i]);

  printf("\n");

  free(a);

  return 0;
}
```

Figura 3: Codi amb fread (veure codi `exemple3.c`).

```c
#include <stdio.h>

#define MAXLINE 100

int main(void)
{
  char buf[MAXLINE];
  while (fscanf(stdin, "%s", buf) != EOF)
    fprintf(stdout, "%s\n", buf);

  return 0;
}
```

Figura 4: Codi que llegeix de l'entrada estàndard amb *scanf* (codi `exemple4.c`).

utilitza la funció *fgets*). Aparentment tots dos programes fan el mateix. Creieu que hi ha un programa més adient que l'altre ? Tindreu la resposta a la secció 2. En tot cas, avancem que per temes de seguretat és molt millor (molt!) fer servir la funció *fgets*.

A la figura 5 es mostra un exemple en què escrivim un vector de *double* a un fitxer. Compareu aquest darrer exemple amb el corresponent que utilitza *fwrite*. Observeu que en aquest cas estem escrivint les dades en format text (les podeu visualitzar amb un editor de text), i que la mida del fitxer és més gran que abans.

A la figura tenim l'exemple que llegeix el fitxer (codi `exemple6.c`).

De la mateixa manera que amb *fread*, la funció *fscanf* ens retorna el nombre d'elements que han estat llegits correctament. Per fer el programa realment segur, hem de comprovar sempre que les dades han estat llegides correctament.

2 Buffer Overflow

Aquest és el nom tècnic que rep, en anglès, el problema que anem a estudiar a continuació. Primer de tot, analitzem els següents dos codis que hem vist anteriorment. Recordem els exemples de les figures 4 (fa servir *ffscanf*) i 1 (fa servir *fgets*).

Aparentment, tots dos codis tenen funcionalitats semblants. Hi ha, però, una diferència important entre dos dos codis: el primer codi no realitza cap comprovació sobre la longitud de les dades que s'introdueixen per teclat, mentre que el segon agafa, a tot estirar, 99 caràcters de l'entrada[1].

Quina importància té això ? Molta! Suposem que estem fent un programa que agafa l'entrada per teclat tal com es mostra a l'exemple 1. Què passarà si introduïm per teclat una cadena de longitud superior a 100 ? Tal com hem dit, amb el codi del primer exemple no es realitza cap comprovació sobre la longitud de la cadena de caràcters que hem entrat. La funció *fscanf* guardarà a partir de la direcció apuntada per *buf* tota la cadena introduïda. Nosaltres només hem reservat espai per 100 caràcters. Per tant, tots els caràcters addicionals (per sobre dels 100) s'escriuran fora d'aquest vector de caràcters. Podem sobreescriure, per exemple, els valors d'altres variables, fent que més endavant es produeixi un *Segmentation fault*. I això que el nostre programa és "correcte". A aquest problema se l'anomena *buffer overflow*.

Vegeu el codi de la figura 7, és un codi vulnerable al *buffer overlow*. Proveu de compilar

[1]La funció *fgets* llegeix $100 - 1 = 99$ caràcters per assegurar que a la variable *buf* hi pugui cabre el caràcter de finalització de cadena '\0' (el byte 0). En cas que la cadena entrada per teclat sigui més llarga de 99 caràcters, la trunca a una longitud de 99. La propera vegada que es faci un *fgets* s'emmagatzemaran a *buf* els caràcters no llegits a la instrucció anterior.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char **argv)
{
  FILE *fp;
  int i, size;
  double *a;

  size = 10;
  a = malloc(sizeof(double) * size);

  for(i = 0; i < size; i++)
    a[i] = sqrt((double) i);

  fp = fopen("vector.data", "w");

  if (!fp) {
    printf("ERROR: no he pogut obrir el fitxer.\n");
    exit(EXIT_FAILURE);
  }

  fprintf(fp, "%d\n", size);

  for (i = 0; i < size; i++)
    fprintf(fp, "%e\n", a[i]);

  fclose(fp);

  return 0;
}
```

Figura 5: Codi per escriure un vector a disc amb fprintf (codi `exemple5.c`).

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  FILE *fp;
  int i, nelems, size;
  double *a;

  fp = fopen("vector.data", "r");

  if (!fp) {
    printf("ERROR: no he pogut obrir el fitxer.\n");
    exit(EXIT_FAILURE);
  }

  nelems = fscanf(fp, "%d", &size);

  if (nelems != 1) {
    printf("ERROR: el fitxer és massa curt.\n");
    exit(EXIT_FAILURE);
  }

  a = malloc(sizeof(double) * size);

  printf("Hi ha %d valors guardats al fitxer.\n", size);
  printf("Els valors son: ");

  for (i = 0; i < 10; i++)
  {
    nelems = fscanf(fp, "%le", a+i); // &(a[i])

    if (nelems != 1) {
      printf("ERROR: el fitxer és massa curt.\n");
      exit(EXIT_FAILURE);
    }

    printf("%f, ", a[i]);
  }

  printf("\n");

  fclose(fp);
  free(a);

  return 0;
}
```

Figura 6: Codi per llegir les dades amb fscanf (codi `exemple6.c`).

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXLEN 10

void main(int argc, char **argv)
{
        char str[MAXLEN];

        if (argc != 2) {
                printf("Usage: %s cadena\n", argv[0]);
                exit(1);
        }

        // Copiem argv[1] a str
        strcpy(str, argv[1]);

        printf("%s\n", str);
}
```

Figura 7: Codi vulnerable al *buffer overlow*, codi `exemple7.c`.

aquest programa, i executar-lo de forma que es produeixi un *Segmentation fault*. Ho aconseguiu ? El problema està en la utilització de la funció *strcpy* per copiar la cadena *argv[1]* (entrada per usuari) a la cadena *str*, que només té reservats 10 caràcters. Per evitar problemes de buffer overflow hauríem d'haver fet servir la funció *strncpy* per copiar la cadena de *argv[1]* a *str*. O bé hauríem pogut fer servir memòria dinàmica per reservar la memòria necessària per *str*.

A l'article que teniu al campus, "An Overview and Example of the Buffer Overflow Exploit", se us explica com els *hackers* utilitzen els *buffer overflows* d'una aplicació per prendre control de l'ordinador. En particular, se us explica com a la cadena introduïda per teclat s'inclouen instruccions màquina (codificats amb bytes) que permeten obrir un terminal en mode superusuari. És molt interessant que li doneu un cop d'ull!

Aquest problema (*buffer overlow*) succeïx quan entrem dades per cadenes de caràcters. S'ha d'anar molt en compte amb funcions utilitzades per manipular les cadenes de caràcters que són entrades al nostre programa (vinguin per línia d'arguments, per teclat, per fitxer, per xarxa, etc.). En el nostre context, a l'hora de realitzar la pràctica, hem d'anar amb compte a l'hora de llegir les dades de disc. És per això que cal que utilitzeu el codi 2 en comptes de 1 perquè l'usuari pugui introduir l'expressió a avaluar. Observeu que la funció *fgets* permet especificar el nombre màxim de caràcters a agafar, mentre que la funció *gets* (la funció homòloga a *fgets* que agafa les dades de *stdin*) no ho permet fer. Per tant, *gets* no és segura en front a *buffer overlow* mentre que *fgets* sí que ho és. Observeu que un cop hem agafat de forma segura les dades de teclat (o de fitxer, etc.) ja podem utilitzar les funcions habituals de manipulació per analitzar la cadena ja que sabem que la longitud màxima està delimitada.

Aquest problema no apareix en llenguatges com el *Java* o el *Python*, ja que en entrar dades per teclat ho fan de forma similar a l'exemple amb *fgets*, que és més segur. Però recordeu que la màquina virtual de *Java* està programada en C (segurament), de la mateixa forma que també ho està l'intèrpret de *Python*. I no és senzill fer que aquestes aplicacions siguin completament segures en front a *buffer overflows*. Altres aplicacions on apareixen usualment problemes de *buffer overflow* són els navegadors web, els servidors web, les aplicacions d'ofimàtica, etc. Totes elles són aplicacions C que són vulnerables a aquest

problema.

3 Bibliografia

1. I. Gerg, "An overview and example of the buffer overflow exploit", IANewsletter (document adjuntat al campus).

2. Rochkind, M.J. Advanced Unix Programming. Addison Wesley, 2004. Aquest llibre està disponible a la biblioteca.

3. Stevens, W.R. Advanced Programming in the Unix Environment. Addison Wesley, 2005.