Sistemes Operatius 1

Sessió 10 - Maig del 2022

Introducció

En aquest document es repassaran diversos aspectes amb què s'ha d'anar amb compte a l'hora de programar en C. En particular, es faran experiments amb dos aspectes de la programació en C que acostumen a portar dificultats:

1. La **pila** és una zona de memòria en què s'emmagatzemen les variables locals declarades a una funció. La pila és una regió de memòria relativament petita i s'ha d'evitar fer servir la pila per emmagatzemar-hi vectors estàtics que tinguin una mida relativament "gran". Què recomanem fer per evitar dificultats a l'hora de programar?
2. Un dels típics problemes de la programació a C és l'**accés als elements d'un vector**, sigui estàtic o dinàmic. El llenguatge C no realitza cap comprovació si l'accés és vàlid com ho fan altres llenguatges, com Python o Java. Això pot provocar que el programa "peti" en una línia de codi diferent de la qual s'ha produït el problema. Com es poden detectar aquests problemes?

Mida de la pila

Comencem per intentar analitzar la mida de la pila en un procés. En concret es proposa realitzar una prova amb el codi *recursive.c*.

```
void funcio(int a)
{
  int b[1000];
  printf("%d\n", a);
  funcio(a+1);
}

int main(void)
{
  funcio(0);

  return 0;
}
```

Observar que el codi és una funció recursiva en què, a cada iteració, "reserva" memòria per a vector estàtic de sencers (cada sencer ocupa 4 bytes de memòria). Executeu el codi i observeu la iteració en què el programa peta.

Pregunta: A la pila s'emmagatzema, a cada crida recursiva, com a mínim el paràmetre a, el vector b i l'adreça de retorn de la funció (8 bytes). Quina és la mida, aproximadament, màxima que un sistema operatiu assigna a la pila?

Anem a fer un experiment per mostrar el fet que reservar espais de memòria grans a la pila pot produir problemes inesperats i que poden ser difícils de resoldre si no s'és conscient de la mida de la pila (són coses que han passat a companys i companyes vostres!).

Analitzeu el codi *proves_mida_pila.c*.

```
#define NCOLS 256

void funcio(int nrows)
{
  char matriu[nrows][NCOLS];

  printf("Faig assignacio i operacio\n");
  // Aixo es fa perque el compilador no es queixi
  // que no es fa servir la variable matriu
  matriu[0][0] = 0;
  matriu[0][0] = matriu[0][0] + 1;
  printf("Surto assignacio i he fet operacio\n");
}

int main(int argc, char **argv)
{
  int nrows;

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

  nrows = atoi(argv[1]);

  printf("Valor de nrows: %d\n", nrows);
  printf("Crido a funcio\n");

  funcio(nrows);

  printf("Surto de funcio, tot ha anat be!\n");

  return 0;
}
```

Observar que el codi reserva una matriu estàtica. En executar el codi es pot passar com a argument el nombre de files de la matriu.

Proveu d'executar el codi amb diferents valors (positius) de *nrow*: 10, 100, 1000, 10.000, 20.000, 30.000, 40.000 i així successivament. Aneu augmentant el valor fins que vegeu que l'aplicació peta.

Pregunta: A partir de quin valor de *nrow*, aproximadament, peta l'aplicació? Quina és la mida en bytes corresponent?

Aquest experiment demostra que la pila és relativament petita. Per això es recomana que, en general, fer servir memòria dinàmica en comptes de memòria estàtica si es volen fer servir vectors (o matrius) de mida gran.

Límits dels vectors

En el llenguatge C es poden fer servir vectors, siguin estàtics (els que s'emmagatzemen a la pila) o dinàmics (la memòria del quals es reserva amb malloc). Una característica particular del llenguatge C és que no es comprova, en temps d'execució, si els accessos que es realitzen a vectors són vàlids. Vegem un exemple, codi *vector_estatic.c*.

```
int main(void)
{
  int a[10];

  printf("Faig assignacions\n");

  a[100] = 1234;
  printf("a[100] = %d\n", a[100]);

  a[1000] = 4321;
  printf("a[1000] = %d\n", a[1000]);

  printf("Surto del main\n");

  return  0;
}
```

Observar que pel vector estàtic només s'ha reservat memòria per a 10 sencers. Al codi es pot veure que s'accedeixen a elements fora de vector. Què passa en executar el codi?

Amb l'exemple anterior hem vist que el llenguatge C no avisa si escrivim fora del vector. La pregunta que ens podem fer és: on estem escrivint? Què estem sobreescrivint? Per això es proposen dos experiments. Aquí teniu el primer codi, *vector_estatic_funcio.c*

```
void funcio()
{
  int a[1000];
  int b;

  b = 5;

  printf("Direccio de b: %x\n", &b);
  printf("Valor de b: %d\n", b);

  printf("Direcció de a[0]: %x\n", &a[0]);
  printf("Direccio de a[-1]: %x\n", &a[-1]);

  a[-1] = 10;

  printf("Valor de b: %d\n", b);
}


int main(void)
{
  funcio();
}
```

Es demana compilar el codi (sense opcions d'optimització) i compilar el codi. Observeu el que passa... en modificar a[-1], estem modificant el valor de b!

Pregunta: per què, en modificar a[-1], estem modificant el valor de b? Feu-vos un dibuix de la pila per veure-ho clar.

Un dels problemes associats a fer servir vectors és que, en sobreescriure un valor fora del vector, podem modificar els valors d'altres variables. Això pot passar tant amb vectors estàtics com dinàmics.

Anem una mica més lluny amb el següent exemple. Sabem que a la pila s'emmagatzema l'adreça de retorn de la funció (és a dir, el punt on ha de continuar l'execució en retornar de la funció). Anem a intentar sobreescriure-la per tal de fer petar el programa! Aquí teniu l'exemple, *codi vector_estatic_funcio2.c.*

```
int funcio(void)
{
  int a[10];
  int i;

  printf("Entro al bucle\n");

  for(i = 12; i < 20; i++)
  {
    a[i] = 2 * i;
    printf("a[%d] = %d\n", i, a[i]);
  }

  printf("Surto del bucle\n");

  return  0;
}

int main(void)
{
  printf("Entro a la funcio\n");
  funcio();
  printf("Surto de la funcio\n");
}
```

Proveu d'executar el codi: al bucle que hi ha estem sobreescrivint informació "crítica" que hi ha a al pila (entre altres coses l'adreça de retorn de la funció). El codi peta en sortir de la funció, no pas en fer el bucle.

En conclusió, aquests exemples mostren que fer servir vectors estàtics poden donar problemes d'execució en el llenguatge C. A més, aquests problemes són difícils de trobar i arreglar. Per això es recomana fer servir, en general, vectors dinàmics per emmagatzemar dades en un vector. Tot i que el llenguatge C tampoc dóna un error en cas que accedim fora del vector, existeixen a l'actualitat eines que permeten detectar i trobar ràpidament els accessos invàlids que es realitzen en fer servir vectors dinàmics. Vegem-ho...

Vectors dinàmics

Comencem amb un exemple similar als anteriors en què demostrem que C no avisa, en temps d'execució, si accedim (sigui per lectura o escriptura) fora d'un vector ubicat de forma dinàmica. Farem servir la funció malloc, que és una crida a llibreria d'usuari. Internament, la funció malloc utilitza la funció sbrk, que és la crida a sistema per demanar memòria dinàmica. La funció malloc intenta "minimitzar" el nombre de crides a sbrk per evitar les crides a funciona.

Vegem el codi *vector_dinamic.c*.

```
int main(void)
{
  int i;
  int *a;

  a = malloc(10 * sizeof(int));

  i = 0;
  while (1)
  {
    a[i] = 2 * i;
    printf("a[%d] = %d\n", i, a[i]);
    i++;
  }

  free(a);

  return 0;
}
```

Executeu aquest codi i observareu que el codi pot escriure fora de la zona de memòria dinàmica que hem reservat. Nosaltres només hem demanat 40 bytes de memòria. Internament, però, la funció malloc reserva una zona de memòria més gran que els 40 bytes que es demanen. La funció malloc demana una zona "gran" i gestiona els espais lliures amb tècniques similars les què esteu fent servir a la pràctica 4.

En el nostre exemple, hem demanat només 40 bytes. Malloc n'ha demanat "molt més" que 40 bytes al sistema operatiu. Atès que aquesta zona ha estat assignada al procés, s'hi pot escriure.

Una de les avantatges de fer servir memòria dinàmica és que aquests tipus d'errors d'accés es poden detectar fàcilment amb eines específiques. Una d'aquestes eines és el ***valgrind*** (que està instal·lat a la màquina virtual). Aquesta eina es farà servir a Sistemes Operatius 2 i no és necessària en aquesta assignatura, tot i que aquí en farem alguns experiments.

Més exemples

Es presenten a continuació altres exemples que han sorgit al llarg dels cursos de Sistemes Operatius 1 arran d'algunes preguntes de companys vostres. Són codis no vàlids, però que donen resultats sorprenents.

Executeu els codis i aviam si podeu explicar el per què del comportament del codi. Aquest és el primer codi, vector_dinamic_free.c

```
int main(void)
{
  int i;
  int *a;

  a = malloc(100 * sizeof(int));

  for(i = 0; i < 100; i++)
    a[i] = 2 * i;

  free(a);

  printf("a[50] = %d\n", a[50]);

  return 0;
}
```

Pregunta: Observeu que s'allibera la memòria del vector i després s'imprimeix el valor d'a[50]. Podeu explicar per què s'imprimeix el valor correcte?

Aquí teniu un exemple similar, vector_dinamic_free2.c.

```
int main(void)
{
  int i;
  int *a, *b;

  a = malloc(100 * sizeof(int));

  for(i = 0; i < 100; i++)
    a[i] = 2 * i;

  free(a);

  b = malloc(1000 * sizeof(int));

  b[50] = 1000;
  printf("a[50] = %d\n", a[50]);

  free(b);

  return 0;
}
```

Pregunta: observeu que en el codi modifiquem el valor de b[50], i després imprimim el valor de a[50]. Podeu explicar el que succeeix?

Exercici a entregar

Com s'ha pogut veure el llenguatge C no realitza, en temps d'execució, comprovacions sobre si l'accés que es fa és vàlid. Té l'avantatge que les aplicacions en C acostumen a ser més ràpides que les equivalents en altres llenguatges. Hi ha l'inconvenient que és més difícil rastrejar els possibles problemes d'accés a memòria que tingui la nostra aplicació.

A l'actualitat existeixen eines que ens faciliten la feina per tal de trobar aquests problemes. A Linux una eina coneguda és l'aplicació *valgrind*. Intuïtivament, aquesta aplicació implementa la seva pròpia versió de malloc cosa que permet detectar els problemes d'accés a memòria i un munt de coses més.

L'objectiu en aquest exercici és veure si l'aplicació *valgrind* es capaç de trobar els problemes que hi ha a les nostres aplicacions. Com es fa servir?

Per obtenir informació sobre els problemes cal

1. Compilar l'aplicació en mode "debugger". Per això cal fer servir l'**opció "-g"** a l'hora de compilar i no pas "-O" que és l'opció que indica que es vol optimitzar el codi.
2. Executar l'aplicació *valgrind* passant-li com a argument l'aplicació compilada.

Vegem un exemple (codi inclòs al directori "exercici")

```
$ gcc -g codi_vector1.c -o codi
$ valgrind ./codi
```

Observeu quina informació ens dóna el *valgrind* en executar l'aplicació. Us ajuda a detectar on és el problema?

Repetiu l'experiment els codis codi_vector2.c, codi_vector3.c, codi_vector4.c, codi_vector5.c.

Avaluació

Es demana entregar un **petit informe de màxim *tres* pàgines** (sense incloure la portada/índex en cas que els vulgueu incloure) que inclogui les proves que heu fet executant el valgrind amb els codis de l'exercici. Inclogueu, per a cada codi, el missatge d'error que us dóna el *valgrind* així com la conclusió que traieu de l'experiment. En particular, on és el problema? Quina informació us dóna valgrind al respecte?

Sigueu breus i concisos a l'hora de comentar quina informació us dóna el *valgrind* respecte el problema que *valgrind* us indica que té el codi.

Entregueu un únic informe en format PDF per grup (grupXX.pdf, on XX és el nombre del vostre grup de pràctiques). L'avaluació per part del professor serà de 0, 2.5, 5, 7.5 o 10. Per fer aquesta avaluació es faran servir criteris equivalents als que es fan als informes que entregueu a pràctiques. És important doncs que interpreteu i comenteu els resultats que obteniu en executar el codi amb el *valgrind*, i presenteu la informació de forma clara i neta.