Ladění programu a realokace paměti

IZP-cv08

Ing. Jakub Husa Ph.D.

Vysoké Učení Technické v Brně, Fakulta informačních technologií
Božetěchova 1/2. 612 66 Brno - Královo Pole
ihusa@fit.vut.cz

14. listopadu 2024

Ladění programu

Ladění běhu programu

Ladění je postup systematického hledání chyb v programu:

- Pro ladění programu ve vývojovém prostředí používáme debugger.
- Debugger (GDB) umožňuje běh programu krokovat po jednotlivých příkazech, a sledovat při tom hodnoty proměnných a zásobník volání funkcí.

Postup ladění ve VS Code:

- Ve složce se zdrojovým kódem (workspace) vytvoříme konfigurační složku .vscode a uložíme do ní konfigurační soubory tasks.json a launch.json.
- V levém bočním panelu zvolíme menu Run and Debug a v horní části panelu vybereme zda chceme program jen spustit (Run) nebo debugovat (Debug).
- Kliknutím nalevo od čísla řádku který chceme debugovat nastavíme breakpoint.
- Zelenou šipkou (F5) přeložíme (s argumenty definovanými v souboru tasks.json) a spustíme (s argumenty definovanými v souboru launch.json) program.
- Běh programu krokujeme tlačítky v horní části okna – Continue (F5), Step over (F10), Step into (F11), Stop (Shift+F5).
- Hodnoty proměnných (Variables, Watch) a zásobník volání (Call stack) sledujeme v levé části okna.

Sledujte okna Variables a Call Stack při krokování následujícího programu:

```
int maximum(int delka, int pole[])
{
    int max = pole[0];                  //pocatecni hodnota maxima
    for(int i = 1; i < delka; i++)      //pro vsechny prvky pole
        if (max < pole[i])              //pokud najdeme prvek ktery je vetsi
            max = pole[i];              //aktualizujeme hodnotu maxima
    return max;                         //vracime maximum
}

int main()                              //zacatek programu
{
    int poleA[5] = {1, 4, 3, 5, 2};  //vytvarime poleA
    int poleB[4] = {10, 30, 20, 40}; //vytvarime poleB

    printf("Maximum poleA je %i\n", maximum(5, poleA)); //vypis
    printf("Maximum poleB je %i\n", maximum(4, poleB)); //vypis
    return 0;                           //konec programu
}
```

Ladění úniků paměti

Úniky paměti můžeme detekovat v unixu programem Valgrind:

- Valgrind kolem našeho programu vytvoří obálku která mu umožní detekovat jestli náš program před ukončením uvolnil veškerou alokovanou paměť:
- Programy které chceme debugovat překládáme s parametrem `-g` :
  překlad: `gcc -g main.c -o main`
  spuštění: `valgrind --leak-check=full --show-leak-kinds=all ./main`
- Pokud k únikům ani k jiným chybám nedošlo, Valgrind by měl vypsat:
  ```
  All heap blocks were freed -- no leaks are possible
  ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
  ```
- V opačném případě Valgrind vypíše kterou alokaci jsme zapomněli uvolnit:
  ```
  40 bytes in 1 blocks are definitely lost in loss record 1 of 1
  at 0x4C29B0D: malloc (vg_replace_malloc.c:299)
  by 0x400556: main (main.c:8) <- neuvolnena alokace
  ```
  Problém nastal ve funkci main v souboru main.c na řádku 8.

Hlášení no leaks are possible znamená že v daném běhu nedošlo k únikům:

- **NEZNAMENÁ** ale, že k nim nedojde pro jiné argumenty nebo vstupní data!
- Valgind kromě úniků detekuje také neplané přístupy do paměti.
- Ve Windows je detekování úniků paměti složitější.

Na procvičení

Vyzkoušejte si:

- Na serveru merlin.fit.vutbr.cz debugujte program z následujícího slajdu, který má ověřit zda soubor input.txt obsahuje rostoucí posloupnost tří celých čísel.

Program obsahuje celkem 7 chyb:

- Špatná velikost alokované paměti.
- Chybějící kontrola úspěšnosti alokace.
- Chybějící kontrola úspěšnosti otevření souboru.
- Chybné indexy porovnávaných položek.
- Chybějící uvolnění paměti.
- Chybějící uzavření souboru.
- Chybějící kontrola úspěšnosti čtení ze souboru.

Pomocí programu Valgind ověřte, že program funguje bez chyb pro:

- Existující soubor obsahující platná data (30 20 10 nebo 10 20 30).
- Ne-existující soubor (který se nepodaří otevřít).
- Existující soubor obsahující neplatná data (písmena místo čísel).

```c
#include <stdio.h>                          //vstup a vystup
#include <stdlib.h>                         //dynamicka alokace pameti
#include <stdbool.h>                        //pravdivostni hodnoty

int main()                                  //zacatek programu
{
    int* pole = malloc(3);                  //alokujeme pole tri cisel
    FILE* soubor = fopen("input.txt", "r"); //otevirame vstupni soubor
    for(int i = 0; i < 3; i++)              //pro vsechny prvky pole
        fscanf(soubor, "%i", &pole[i]);     //nacitame vstup

    bool rostouci = true;                   //posloupnost je rostouci
    for(int i = 0; i < 3; i++)              //pro vsechny prvky pole
        if (! (pole[i] < pole[i+1]))        //pokud dalsi neni vetsi
            rostouci = false;               //posloupnost neni rostouci

    if (rostouci)                           //pokud jsme nenasli problem
        printf("JE rostouci\n");            //posloupnost JE rostouci
    else                                    //jinak
        printf("NENI rostouci\n");          //posloupnost NENI rostouci
    return 0;                               //konec programu
}
```

Realokace paměti

Realokace paměti

Velikost dynamicky alokované paměti můžeme změnit funkcí realloc z knihovny stdlib.h:

- Vstupem funkce je současná adresa realokovaného pole a jeho nová velikost.
- Výstupem funkce je nová adresa pole.

Dynamicky alokované proměnné **NEJSOU** na hromadě alokovány jako souvislý blok dat:

- Pokud je na současné pozici dostatek místa, relokace pole rozšíří na novou velikost.
- Pokud na současné pozici dostatek místa není, realokace všechny položky překopíruje na novou adresu, a starou adresu uvolní.

Alokovanou paměť můžeme realokací i zmenšit:

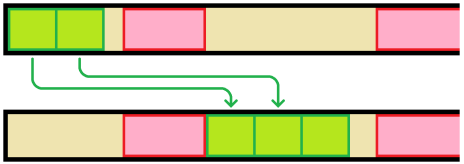
- Při zmenšování může dojít ke změně adresy.

```
1 //alokujeme pole peti cisel
2 int* x = malloc(sizeof(int)*5);
```

Alokovaná, Nedostupná a Volná Paměť

```
3 //realokujeme na deset cisel
4 x = realloc(x, sizeof(int)*10);
```

```
5 //realokujeme na patnact cisel
6 x = realloc(x, sizeof(int)*15);
```



Na vyzkoušení

```c
int* foo() {
    int* a = malloc(sizeof(int)*2); //alokujeme pole dvou cisel
    a[0] = 10;                      //prvni polozka je 10
    a[1] = 20;                      //druha polozka je 20
    return a;                       //vracime ukazatel na alokovane pole
}

int* bar(int* b) {
    int* c = realloc(b,sizeof(int)*3); //pole rozsirime na tri cisla
    c[2] = 30;                      //treti polozka je 30
    return c;                       //vracime ukazatel na realokovane pole
}

int main() {                        //zacatek programu
    int* x = foo();                 //pole vytvorene funkci foo
    x = bar(x);                     //a protazene funkci bar
    printf("x[0] = %i\n", x[0]);    //vypis
    printf("x[1] = %i\n", x[1]);    //vypis
    printf("x[2] = %i\n", x[2]);    //vypis
    free(x);                        //uvolnujeme pamet
    return 0;                       //konec programu
}
```

Ošetření realokace

Pokud realokace selže, funkce vrátí NULL, ale původní adresu **NEUVOLNÍ**:

- Pokud jsme ukazatel již přepsali, adresu nepůjde uvolnit a nastává únik paměti.

```
1 int* x = malloc(sizeof(int) * 5);          //tuto pamet nepujde uvolnit
2 x = realloc(x, sizeof(int) * 10);          //pokud tato realokace selze
```

Uniklá paměť zůstane nepoužitelná dokud nedojde k ukončení celého programu:

- Realokaci paměti můžeme ošetřit použitím pomocného ukazatele.

```
3 int* y = realloc(x, sizeof(int) * 15);  //realokujeme pomocny ukazatel
4 if (y != NULL)                          //pokud realokace uspela
5     x = y;                              //aktualizujeme puvodni ukazatel
6 else                                    //jinak (pokud realokace selhala)
7     free(x);                            //uvolnujeme starou adresu
```

Realokace prázdného ukazatele (NULL) je v pořádku – provede se malloc:

- Realokace na velikost 0 je chyba – provede se free, ale Valgind si bude stěžovat.

```
8 int* z = realloc(NULL, sizeof(int)*20); //realokace vytvori nove pole
9 z = realloc(z, 0);                      //realokace na velikost 0 - CHYBA
```

Sledujte jak se realokací bude měnit velikost a adresa alokovaného pole:

```
int main()                              //zacatek programu
{
    int velikost = 1;                   //velikost alokovane pameti
    void* pole = malloc(velikost);      //alokujeme pole (obecneho typu)
    while (pole != NULL)                //dokud alokace neselhala
    {
        printf("Pole zabira %10u B na adrese %10u\n", velikost, pole);
        velikost *= 2;                  //zvysujeme velikost

        void* temp = realloc(pole, velikost); //realokujeme pole
        if (temp != NULL)               //pokud realokace uspela
            pole = temp;                //aktualizujeme adresu pole
        else                            //jinak (pokud realokace selhala)
        {
            free(pole);                 //starou adresu pole uvolnujeme
            pole = NULL;                //zneplatnujeme ukazatel
        }
    }
    return 0;                           //konec programu
}
```

Na procvičení

Vyzkoušejte si:

- Implementujte následující funkce pro práci s datovým typem vektor.

```
1  void vypisVektor(vektor* A);
2  void pridejPosledni(vektor* A, int cislo);
3  void odeberPosledni(vektor* A);
4  void odeberVsechny(vektor* A);
5  void pridejPrvni(vektor* A, int cislo);
6  void odeberIndex(vektor* A, int index);
```

- vypisVektor – vypíše hodnotu všech prvků pole vektoru.
- pridejPosledni – realokuje pole a na konec přidá nový prvek.
- odeberPosledni – ověří zda je pole prázdné, a pokud ne, tak z něj realokací odebere poslední prvek.
- odeberVsechny – z pole smaže všechny prvky a uvolní ho.
- pridejPrvni – realokuje pole a na začátek přidá nový prvek.
- odeberIndex – ověří zda prvek se zadaným indexem existuje, a pokud ano, tak ho smaže a provede realokaci.

- Na serveru merlin.fit.vutbr.cz si ověřte že v programu nedochází k únikům paměti ani k jiným chybám.

Například:

```
1  Vektor:
2  Vektor: 10
3  Vektor: 10 20
4  Vektor: 10 20 30
5  Vektor: 10 20
6  Vektor:
7  Vektor:
8  Vektor: 40
9  Vektor: 50 40
10 Vektor: 60 50 40
11 Vektor: 60 50 40
12 Vektor: 60 50 40
13 Vektor: 60 40
14 Vektor:
```

```
typedef struct Svektor                  //deklarujeme datovy typ pro strukturu
{                                       //jmenem "Svektor" se dvema polozkami
    int delka;                          //pocet polozek
    int* pole;                          //dynamicky alokovane pole
} vektor;                               //jmeno tohoto typu je "vektor"

int main()                              //zacatek programu
{   //"A" je automaticky alokovany vektor obsahujici dynamicky alokovane pole
    vektor A = {0,  NULL};  vypisVektor(&A); //vytvarime vektor
    pridejPosledni(&A, 10); vypisVektor(&A); //pridavame posledni prvek
    pridejPosledni(&A, 20); vypisVektor(&A); //pridavame posledni prvek
    pridejPosledni(&A, 30); vypisVektor(&A); //pridavame posledni prvek
    odeberPosledni(&A);     vypisVektor(&A); //odebirame posledni prvek
    odeberVsechny(&A);      vypisVektor(&A); //odebirame vsechny prvky
    odeberPosledni(&A);     vypisVektor(&A); //odebirame neexistujici prvek
    pridejPrvni(&A, 40);    vypisVektor(&A); //pridavame prvni prvek
    pridejPrvni(&A, 50);    vypisVektor(&A); //pridavame prvni prvek
    pridejPrvni(&A, 60);    vypisVektor(&A); //pridavame prvni prvek
    odeberIndex(&A, -1);    vypisVektor(&A); //odebirame neexistujici prvek
    odeberIndex(&A, 3);     vypisVektor(&A); //odebirame neexistujici prvek
    odeberIndex(&A, 1);     vypisVektor(&A); //odebirame konkretni prvek
    odeberVsechny(&A);      vypisVektor(&A); //odebirame vsechny prvky
    return 0;                           //konec programu
}
```