Numele si prenumele (cu MAJUSCULE): _____ Grupa: _____

Test L3: _____ Tema: _____ Colocviu: _____ FINAL: _____

Test de laborator - Arhitectura Sistemelor de Calcul

17 ianuarie 2023

Varianta 2

- Nota maxima pe care o puteti obtine este 10.
- Nota obtinuta trebuie sa fie minim 5 pentru a promova, fara nicio rotunjire superioara.
- Orice tentativa de frauda este considerata o incalcare a Regulamentului de Etica!

1 Partea 0x00 - maxim 4p

Consideram ca vrem sa aplicam un algoritm de *clustering* intr-un graf ponderat (in care muchiile au costuri asociate). Algoritmul va grupa intr-un *cluster* (componenta conexa) acele varfuri care sunt cele mai apropiate din punctul de vedere al costului muchiilor dintre ele, fara sa se depaseasca o anumita valoare prestabilita. De exemplu, daca distanta maxima pentru a putea considera doua varfuri intr-o componenta conexa este 10, si intre varfurile `0` si `2` avem o muchie de cost 15, nu putem pune varfurile `0` si `2` in aceeasi componenta. Pentru a rezolva aceasta problema, se considera o varianta modificata a algoritmului Kruskal, care obtine arborele partial de cost minim. In contextul dat, impunand conditiile de mai sus, vom obtine mai multi arbori partiali de cost minim.

Consideram ca avem implementata o procedura `(long*, long) Kruskal(long *matrix)` care primeste ca intrare o matrice de costuri si returneaza: 1. un *array* de elemente de tip `.long` care, pe fiecare pozitie `i` contine *clusterul* in care se afla nodul cu indexul `i`; si 2. dimensiunea acestui *array*. De exemplu, putem obtine ca retur din procedura `([0 1 0 2 0 3 1 2], 8)`, ceea ce inseamna ca nodul 0 a ajuns in componenta conexa 0, nodul 1 in componenta conexa 1, nodul 2 in componenta conexa 0 etc.

Subiectul 1 (3p) Implementati o procedura `long countFirstCluster(long *matrix)` care primeste ca argument matricea de costuri si returneaza numarul de noduri aflate in primul cluster in urma aplicarii algoritmului Kruskal modificat. In `countFirstCluster` vom avea un apel intern cate procedura `Kruskal`. Se garanteaza ca exista noduri in clusterul 0. (in exemplul anterior, sunt 3 noduri in acest cluster) **Incepeti sa scrieti rezolvarea de pe aceasta pagina, si apoi folositi caseta de pe pagina urmatoare, in cazul in care mai aveti nevoie de spatiu**

Solution:

```
countFirstClusters:
    push %ebp
    movl %esp, %ebp

    movl 8(%ebp), %eax

    push %eax
    call Kruskal
    addl $4, %esp

    push %ebx
    push %edi

    movl %eax, %edi
    xorl %eax, %eax
    xorl %edx, %edx

et_for:
    cmp %edx, %ecx
    je et_exit
    movl (%edi, %edx, 4), %ebx
    cmpl $0, %ebx
    je et_count

et_cont:
    incl %edx
    jmp et_for

et_count:
    incl %eax
    jmp et_cont


et_exit:
    pop %edi
    pop %ebx
    pop %ebp
    ret
```

Subiectul 2 (1p) Sa se reprezinte configuratia stivei in momentul in care are adancimea maxima, pentru procedura `countFirstCluster`. Stim ca procedura `Kruskal` respecta toate conventiile de apel si utilizeaza intern toti registrii cu exceptia lui %esi, precum si 4 variabile locale.

Solution: Solutia pentru procedura propusa (depinde de implementare!):

```
%esp:
    (<variabila locala 4 Kruskal>)
    (<variabila locala 3 Kruskal>)
    (<variabila locala 2 Kruskal>)
    (<variabila locala 1 Kruskal>)
    (%edi Kruskal)
    (%ebx Kruskal)
%ebpKruskal:
    (%ebp Kruskal)
    (<r.a. Kruskal>)
    (*matrix)
%ebpClusters:
    (%ebp countClusters)
    (<r.a.countClusters>)
    (*matrix)
```

2 Partea 0x01 - maxim 3.5p

Subiectul 1 (0.7p) Fie urmatoarea secventa de cod:

```
et0:
    movl $0x10, %eax
    shr $5, %eax
    decl %eax
    xorl %edx, %edx
    cmpl %eax, %edx
    ja et1
    jmp et2
et1:
    movl $1, %ebx
    jmp et3
et2:
    movl $2, %ebx
et3:
```

Executam, in *debugger*, secvential, urmatoarele comenzi:

a. `b et0; run; stepi; stepi; i r eax`

b. `b et0; b et3; run; stepi; c; i r ebx`

Scrieti valoarea afisata la finalul fiecarui set de comenzi.

Solution: a. in eax se gaseste valoarea 0, este shiftare la dreapta cu 5, deci impartire cu $2^5 = 32$; b. ni se cere valoarea lui %ebx la eticheta `et3`, asa ca vom continua rationamentul programului: dupa ce %eax devine 0, este decrementat, si este comparat cu %edx, care este egal cu 0. Comparatia utilizata este pe **unsigned**, astfel ca se compara `0` cu `0xffffffff`. Evident, `0` este mai mic, si se face astfel salt la et2, unde %ebx primeste valoarea 2.

Subiectul 2 (0.7p) Fie urmatorul sir de instructiuni:

```
mov $0x20000000, %eax
mov $0x40, %ebx
mul %ebx
mov $0x10, %ecx
div %ecx
```

Ce valori vom gasi in registrii `%eax`, respectiv `%edx` dupa ce executam cele 5 instructiuni? Ce s-ar intampla daca am inlocui instructiunea `mov $0x10, %ecx` cu `mov $0x2, %ecx`?

Solution: Executam codul pas cu pas. Valoarea `$0x20000000` este $2 \cdot 16^7 = 2 \cdot 2^{28} = 2^{29}$, iar in urma executarii urmatoarei linii, vom gasi ca `%eax` $= 2^{29} \cdot 2^6$ (`$0x40` $= 4 \cdot 16 = 2^6$), deci `%eax` $= 2^{35} = 2^{32} \cdot 2^3$, de unde aflam ca `%eax = 0` si `%edx = 8` pana in momentul efectuarii impartirii. Continuam cu impartirea, avem valoarea `0x10 = 16` pusa in %ecx (deci 2^4), iar `div %ecx` va produce `(%edx, %eax) = (%edx, %eax) / %ecx`, deci `(%edx, %eax)` $= (0, \frac{2^{35}}{2^4}) = (0, 2^{31})$ (0 pe prima pozitie pentru ca impartirea este exacta, restul obtinut este 0), ceea ce inseamna in reprezentare binara pentru cat `100...00`, `1` urmat de 31 de `0`-uri, adica `0x80000000`. Obtinem ca `%edx` este 0, iar `%eax` este `0x80000000`. Daca incercam sa impartim doar la 2, rezultatul este 2^{33} (rest 0), dar catul nu este reprezentabil pe 32 de biti, astfel ca vom obtine o exceptie aritmetica.

Subiectul 3 (0.7p) Justificati posibilitatea aparitiei erorii `Segmentation fault` in momentul in care efectuati un apel `scanf`, dar omiteti utilizarea simbolului `$` pentru argumentele de dupa sirul de format. (de exemplu, `push x; push $fmt; call scanf; addl $8, %esp` in loc de `push $x; push $fmt; call scanf; addl $8, %esp`)

Solution: Procedura `scanf` scrie valoarea introdusa la standard input la adresa de memorie primita ca argument. Daca omitem simbolul $, cel mai probabil se va accesa o zona de memorie in care programul nostru nu are drepturi.

Subiectul 4 (0.7p) Consideram ca avem functia `g` cu doua argumente de tip `.long`, care utilizeaza local registrii `%eax`, `%ebx`, `%ecx` si `%edi`, precum si 8 variabile locale de tip `.long`. Aceasta functie este **recursiva** si respecta, in implementare, toate conventiile pentru crearea cadrului de apel prezentate la laborator. Stiind ca spatiul stivei programului vostru variaza intre `0xdedf5000` (inferior) si `0xdedf3000` (superior), dupa cate **autoapeluri** se va depasi spatiul alocat stivei?

Solution: Se determina mai intai spatiul de care dispunem, si anume diferenta dintre `0xdedf5000` si `0xdedf3000`, care este `0x2000`, ceea ce in zecimal inseamna $2 \cdot 16^3 = 8192$. Aceasta valoare este exprimata in Bytes. Determinam acum cat ocupa, in Bytes, stiva locala pentru cadrul de apel. Pentru procedura noastra intr-o configuratie locala avem cele doua argumente + adresa de retur + salvarea %ebp-ului + salvarea %ebx + salvarea %edi + 8 variabile locale, insemnand 14 spatii, adica 56B. Facem impartirea, iar 8192 / 56 = 146 rest 16, deci la al 147-lea autoapel vom depasi acest spatiu alocat.

Subiectul 5 (0.7p) Fie urmatorul cod scris in limbajul `C`. Descrieti care va fi efectul executarii codului din `main`, utilizand cunostintele de Assembly din acest semestru.

```
void f()
{
    long x = 5, y = 7;
}
void g()
{
    long p, q;
    printf("%d", q);
}
```

```
int main()
{
    f();
    g();

    return 0;
}
```

Solution: Afiseaza 7, pentru ca atat perechile (x, y), cat si (p, q) vor fi acelasi element in stiva, q-ul din g va citi ce exista deja la -8(%ebp), care a fost deja completat cu 7 de y-ul din f.

3 Partea 0x02 - maxim 2.5p

Presupunem ca aveti acces la un executabil `exec`, pe care il inspectati cu `objdump -d exec`. In momentul in care rulati aceasta comanda, va opriti asupra urmatorului fragment de cod. Analizati acest cod si raspundeti la intrebarile de mai jos. Pentru fiecare raspuns in parte, veti preciza si liniile de cod / instructiunile care v-au ajutat in rezolvare.

```
<func>:
     1.    pushl   %ebp
     2.    movl    %esp, %ebp
     3.    subl    $24, %esp
     4.    movl    20(%ebp), %edx
     5.    movl    24(%ebp), %eax
     6.    movb    %dl, -20(%ebp)
     7.    movb    %al, -24(%ebp)
     8.    movl    $0, -4(%ebp)
     9.    movl    $0, -8(%ebp)
.L4:
    10.    movl    -8(%ebp), %eax
    11.    cmpl    12(%ebp), %eax
    12.    jge     .L2
    13.    movl    -8(%ebp), %eax
    14.    cmpl    16(%ebp), %eax
    15.    jl      .L3
    16.    movl    -8(%ebp), %edx
    17.    movl    8(%ebp), %eax
    18    addl    %edx, %eax
    19.    movzbl  (%eax), %eax
    20.    cmpb    %al, -20(%ebp)
    21.    jge     .L3
    22.    movl    -8(%ebp), %edx
    23.    movl    8(%ebp), %eax
    24.    addl    %edx, %eax
    25.    movzbl  (%eax), %eax
    26.    cmpb    %al, -24(%ebp)
    27.    jle     .L3
    28.    addl    $1, -4(%ebp)
.L3:
    29.    addl    $1, -8(%ebp)
    30.    jmp     .L4
.L2:
    31.    movl    -4(%ebp), %eax
    32.    ret
```

a. (0.5p) Cate argumente primeste procedura de mai sus?

Solution: 5 argumente - observam la linia 5 un `24(%ebp)`, deci avem 8, 12, 16, 20, 24 - argumentele procedurii, ca offset relativ la %ebp

b. (0.5p) Ce tip de date returneaza aceasta procedura?

Solution: trebuie sa urmarim registrul `%eax`: ultima lui aparitie este la linia 31, unde se executa un `mov` din locatia `-4(%ebp)`. Urmarim cum folosim acest `-4(%ebp)` si observam, la linia 28, ca se face asupra lui un `addl`, de unde conchidem ca este un `.long`. In concluzie, procedura returneaza un `.long`.

c. (0.5p) In procedura sunt utilizate si variabile locale. Descrieti care este scopul variabilei locale situata la adresa `%ebp - 8`.

Solution: urmarim -8(%ebp) in codul primit. observam ca este initializat cu 0 la linia 9. Ulterior, la eticheta .L4, observam ca -8(%ebp) este pus in %eax, cu scopul de a fi comparat cu 12(%ebp). Pana aici conchidem ca -8(%ebp), si argumentul procedurii 12(%ebp) (cel de-al doilea argument al procedurii) sunt elemente de tip .long, si se compara cu cmpl. Relatia devine, in limbaj natural, daca -8(%ebp) este mai mare sau egal cu 12(%ebp), sare la .L2, unde se face

un return. Pana in acest punct, observam ca -8(%ebp) este utilizat intr-o conditie importanta de finalizare a logicii procedurii. Continuam cu analiza, si observam, in eticheta .L3 ca, inainte de saltul la .L4, se face o incrementare in -8(%ebp). Conchidem, deci, ca -8(%ebp) este un index intr-o structura repetitiva.

d. (0.5p) Stiind ca `movzbl` reprezinta un `mov` cu conversie de tip, de la `.byte` la `.long`, care este tipul de date al primului argument al procedurii?

Solution: Primul argument se afla la 8(%ebp), deci trebuie sa urmarim aparitiile acestuia. Parcurgem liniar codul, si gasim 8(%ebp) la linia 17, mutat in registrul %eax. Observam ca %eax este modificat sa devina %eax + %edx, unde %edx este valoarea indexului curent dintr-o structura repetitiva, conform mov-ului de la linia 16 si conform a ceea ce am determinat anterior. Observam la linia 19 un detaliu important: se face un mov cu conversie, astfel incat %eax sa stocheze continutul de memorie de la adresa din %eax! Daca asamblam informatia, 8(%ebp) reprezinta o adresa de memorie pe care o putem modifica cu orice numar intreg (nu cu multipli de 2, 4 ...), si asupra careia aplicam un `movzbl`, ceea ce ne duce cu gandul la un char* (byte ptr).

e. (0.5p) Descrieti comportamentul procedurii de mai sus, utilizand ceea ce ati descoperit la subpunctele anterioare.

Solution: Urmarim rezultatele anterioare, si stim ca avem 5 argumente, primul un char*, apoi un long, apoi trei argumente pe care nu le-am analizat, dar le-am vazut ca apar in structura repetitiva. Revenim la eticheta .L4 sa continuam analiza codului. Intr-adevar, liniile 10-12 am vazut ca reprezinta iesirea din structura repetitiva, deci 12(%ebp) ar fi o dimensiune maxima de parcurs, cel mai probabil pentru char*-ul primit ca prim argument, stocat la 8(%ebp). Continuam sa citim codul, si consideram liniile 13-15. Se pune indexul curent in %eax, si al treilea argument (16(%ebp) in $eax), si daca indexul este mai mic strict decat argumentul 3, se merge la pasul urmator (la .L3 avem incrementare de contor si revenire la .L4), deci este necesar pentru a continua ca indexul sa fie mai mare sau egal decat argumentul 3. Observam inca doua astfel de salturi, foarte similare, dar cu conditie schimbata - la linai 21 un jge .L3, respectiv la 27 un jle .L3. Analizam liniile 16-21. Se pune indexul in %edx, se pune primul argument, deci adresa sirului in %eax, se adauga %edx la %eax si se acceseaza elementul de la adresa de memorie %eax + %edx, adica elementul curent din sir. Se compara acum acest element cu -20(%ebp), dar de la linia 6 stim ca -20(%ebp) condine ultimul byte din registrul %edx, in care am incarcat la linia 4 al patrulea argument. Astfel, comparam elementul curent cu argumentul 4, iar relatia de ordine spune ca daca argumentul este mai mare sau egal decat elementul curent din sir, mergem la L3, deci o conditie de continuare este ca elementul curent din sir sa fie mai mare strict decat argumentul 4. Aplicand exact aceeasi logica pentru liniile 22-27, obtinem ca elementul curent din sir trebuie sa fie mai mic strict decat argumentul 5. Dupa ce am tratat cazurile de oprire a pasului curent, observam ca singura operatie care ramane in structura repetitiva este un `addl $1, -4(%ebp)` la linia 28, care doar incrementeaza o variabila locala initial facuta 0 la linia 8,

si am vazut la punctul a) ca tocmai aceasta valoare este returnata. In concluzie, procedura returneaza cate elemente din sir respecta cele trei conditii descrie anterior:

```
long func(char *arg1, long arg2, long arg3, char arg4, char arg5)
{
    long result = 0
    long index = 0
    while (index < arg2)
    {
        if (index >= arg3 && arg1[index] > arg4 && arg1[index] < arg5)
        {
            result += 1
        }
        index += 1
    }
    return result
}
```