Tema 2 - Memoria lui Biju

- Deadline HARD: 17.12.2021, ora 23:55
- Data publicării: 22.11.2021
- Responsabili:
  - Ilinca-Ioana Struțu [mailto:ilinca.strutu@gmail.com]
  - Florin Postolache [mailto:florin.postolache@stud.acs.upb.ro]
  - Răzvan-Nicolae Vîrtan [mailto:virtanrazvan@gmail.com]
- Actualizări:
  - 23.11.2021 - 18:30 - little fix în checker [https://github.com/systems-cs-pub-ro/iocla/tree/master/teme/tema-2]
  - 23.11.2021 - 01:00 - checker și teste [https://github.com/systems-cs-pub-ro/iocla/tree/master/teme/tema-2]
  - 23.11.2021 - 10:44 - Deadline-ul este soft
  - 27.11.2021 - 12:30 - Clarificări coding style și enunț task 2
  - 07.12.2021 - 20:00 - Anulare deadline SOFT

Enunț

Acum că știe mai multe despre sistemele de fișiere, Biju vrea să devină expert în memorie, în drumul către răzbunarea sa. Pentru asta, s-a înscris într-o echipă de hackeri, însă a realizat repede că nu e suficient de pregătit, pentru că nu-l întreabă nimeni nimic. Sarcina voastră este să-l ajutați să rezolve câteva task-uri pentru a deveni mai bun și nu în ultimul rând, pentru a deveni un om întrebat.

Structură și detalii de implementare

Tema este formată din 4 exerciții independente. Fiecare task constă în implementarea unei funcții în limbaj de asamblare. Implementarea se realizează în fișierele puse la dispoziție pentru fiecare exercițiu.

Parametrii funcţiilor sunt plasaţi în registre, în cadrul scheletului.

Scheletul include şi macro-ul PRINTF32, folosit în laborator, pentru a vă ajuta la depanarea problemelor. Tema finală nu trebuie să facă afişări folosind PRINTF32, funcții externe sau apeluri de sistem.

În tema finală este interzisă apelarea funcţiilor externe.

IOCLA RULZ!

---

1. Reversed One Time Pad - 10p

Pentru început, Biju se încalzește cu un exercițiu de criptare. Pentru ca a găsit pe internet cum se implementează One Time Pad, Biju s-a gândit să complice puțin algoritmul prin inversarea cheii. Astfel, Biju va efectua operația XOR între mesajul său **plaintext** și cheia inversată **key** , ambele de lungime **len** . Rezultatul funcției este salvat mai apoi în **ciphertext** .

```
ciphertext[i] = plain[i] ^ key[len - i - 1]
```

Biju trebuie să implementeze, în assembly, funcția de criptare Reverse One Time Pad. Antetul funției, în C, este:

```
void rotp(char *ciphertext, char *plaintext, char *key, int len)
```

---

2. Ages - 15p

Acum că și-a reamintit cum se scria cod in assembly, Biju face o pauză și se uită pe un fișier primit de la un coechipier hacker. El realizează ca acest fișier conține zilele de nastere ale coechipierilor săi. Pentru ca lui Biju nu i-a plăcut matematica în liceu și nici la facultate, se hotărăște să scrie o funcție care calculează vârsta coechipierilor săi știind când s-au născut și care este data curentă.

Astfel, Biju își definește o structură în care dorește să păstreze elementele unei date:

```
struct my_date{
    short day;
    short month;
    int year;
} __attribute__((packed));
```

Din acest fișier, Biju extrage numărul de persoane din echipă **len**, data curentă într-o variablia **present** și un vector care cuprinde toate datele de naștere ale coechipierilor săi **all_ages**. Rezultatul acestei funcții este un vector ce conține vârstele coechipierilor **all_ages**.

Antetul funcției, în C, este:

```
void ages(int len, struct my_date* present, struct my_date* dates, int* all_ages)
```

Trebuie să verificați toate cazurile posibile, inclusiv cazul în care data de naștere este după data prezentă, caz în care trebuie returnat 0.

Înainte de rezolvarea acestui task recomandăm parcurgerea laboratorului 7

https://ocw.cs.pub.ro/courses/iocla/laboratoare/laborator-07 [https://ocw.cs.pub.ro/courses/iocla/laboratoare/laborator-07]

CREZI CĂ AI AJUNS LA JUMĂTATE? NOT SO SURE ABOUT THAT!

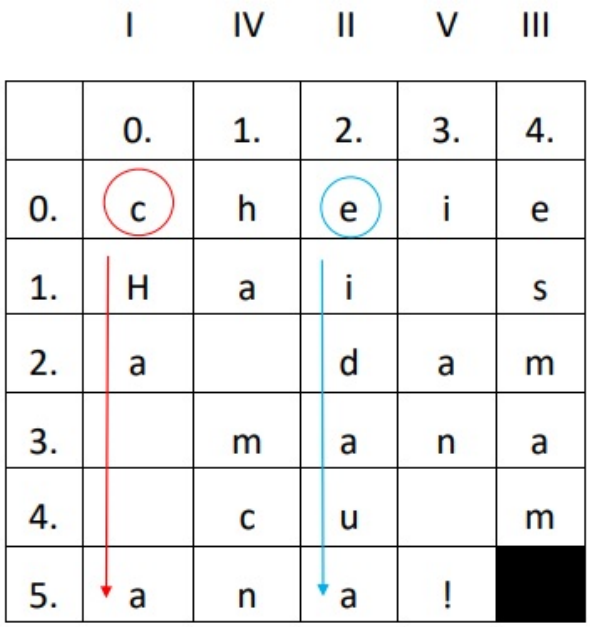
---

3. Columnar Transposition Cipher - 25p

Pentru că momentan a fost întrebat doar cât este ceasul, Biju dorește să învețe tehnici avansate de criptare. (mai avansate decât rotp) Cu acestea, el dorește să cripteze toate documentele viitoarelor sale victime. După un search rapid pe Google, Biju a dat peste cifrul de

transpoziție al coloanelor. Acesta pornește de la o cheie de lungime **l_key** și un plaintext de lungime **l_plain**. Se creeaza o matrice cu **l_key** coloane și **ceil(l_plain/l_key) + 1** [1)] linii. În aceasta, se trece pe prima linie cheia. Pe următoarele linii se trece plaintextul până la terminarea acestuia.

Exemplu:



Textul codat se construiește prin citirea pe coloană a caracterelor din plaintext. Ordinea citirii coloanelor este ordinea lexicografică a caracterelor din cheie. În exemplul de mai sus oridnea coloanelor este: [0, 2, 4, 1, 3].

Exemplu codare:

Text codat:

Dupa pasul I: Ha a

Dupa pasul II: Ha aidaua

Dupa pasul III: Ha aidauasmam

Dupa pasul IV: Ha aidauasmama mcn

Dupa pasul V: Ha aidauasmama mcn an !

Deoarece Biju nu știe încă să sorteze matrici în funcție de coloane, a "împrumutat" de la colegul său de echipă o funcție care transformă cheia în vectorul de ordine al coloanelor descris mai sus. Ceea ce îi mai rămâne lui Biju de făcut este ca, având la îndemână vectorul de ordine si plaintextul, să realizeze codarea textului.

Biju trebuie să scrie funcția columnar_transposition, care primește ca parametrii vectorul de ordine al coloanelor, plaintextul și întoarce în ciphertext rezultatul codării.

Pentru a-și elibera câteva registre, Gigel își salvează lungimea cheii și lungimea plaintextului în variabilele **len_cheie** și **len_haystack** .

Antetul funcției, în C, este:

```
void columnar_transposition(int key[], char *haystack, char *ciphertext)
```

Pentru usurinta, toate cheile vor avea doar litere mici.

---

4. Cache Load Simulation - 40p

AM VENIT SĂ VĂ ÎNVĂȚĂM MESERIE!

Tocmai când și-a încheiat aventura criptografică și a crezut că e deja expert în memorie, Biju a aflat că lucrurile sunt și mai complicate, deoarece pe lângă memoria RAM mai există un tip special de memorie, numită **cache** . Sarcina voastră finală este să creați pentru Biju o simulare simplă de încărcare a datelor din memorie în registre, ținând cont și de existența cache-ului.

După cum știți deja, atunci când trebuie să lucreze cu niște date, procesorul trebuie mai întâi să le aducă din memoria RAM în registre(memoria sa internă). Memoria RAM are avantajul că poate păstra cantități mari de date, însă accesarea datelor de către procesor din memoria RAM durează relativ mult. În schimb, registrele pot fi accesae foarte rapid (fac parte din procesor), însă au o dimensiune redusă și nu putem stoca în ei foarte multe date simultan.

Memoria cache reprezintă o soluție de compromis, utilizată în aproape toate sistemele moderne de calcul. Aceasta este o unitate de memorie adițională, plasată între memoria RAM și procesor. Dimensiunea ei este mai mare decât cea a registrelor, însă mai mică decât cea a RAM-ului. De asemenea, timpul de acces este mai bun decât timpul de acces al datelor din RAM, iar asta ne ajută să scurtăm timpii de aducere a datelor din memoria RAM pe procesor.

Atunci când procesorul are nevoie de date de la o anumită adresă din RAM, acesta caută mai întâi în cache pentru a verifica dacă datele respective există acolo. Dacă da, înseamnă că avem un **CACHE HIT** și putem prelua datele direct din cache, fără a mai interacționa cu memoria principală, deci economisind timp. Dacă datele nu există încă în cache, înseamnă că avem un **CACHE MISS** , situație în care datele trebuie mai întâi aduse din RAM în cache, după care trebuie transferate în registrele procesorului. Deși la un **CACHE MISS** avem un delay suplimentar, a doua oară când vom încerca să accesăm date de la aceeași adresă le vom putea accesa mai rapid, deoarece acum se află în cache (cu condiția să nu fi fost între timp suprascrise!).

Structura cache-ului

- Cache-ul din cadrul acestui task este unul simplificat și are forma unui tabel cu 100 de linii, fiecare linie având 8 octeți. În cadrul implementării, cache-ul este reprezentat ca o matrice alocată static, **char cache[CACHE_LINES][CACHE_LINE_SIZE]** . Pe o linie din cache vom avea, la un moment dat, 8 octeți consecutivi din memorie, adresa primului octet de pe linie fiind mereu divizibilă cu 8 (adică are ultimii 3 biți 0).

- Fiecare linie din `cache` are asociat un tag. Dacă linia respectivă nu conține date, tag-ul e 0. Dacă linia din cache conține date, tag-ul e egal cu valoarea obținută din primii 32 - 3 = 29 de biți ai adresei primului octet de pe linia respectivă. Spre exemplu:

```
adresa primului octet de la linia 12 din cache:  00111001000001001110001000111000
valoarea tag-ului pentru această linie:          00111001000001001110001000111
```

Observăm că am tăiat practic ultimii 3 biți din adresă. Tag-ul este utilizat în căutarea datelor de la o anumită adresă în cache. În cadrul implementării, tag-urile sunt reprezentate ca un vector cu **CACHE_LINES** elemente:

```
char** tags
```

- Pentru o linie de cache care nu conține date, tag-ul este 0.
- "Registrul" în care ne vom dori să încărcăm date va fi de fapt o variabilă de 1 octet, **char reg** . Practic, ne va interesa să primim o adresă de memorie și să încărcăm în final un octet de la acea adresă în **reg** .

Algoritmul de load

Antetul funcției load are următoarea formă:

```
void load(char *reg, char** tags, char **cache, char *address, int to_replace)
```

Pe lângă parametrii **reg** , **tags** și **cache** , deja explicați, **address** reprezintă adresa octetului care trebuie în final adus în **reg** , iar **to_replace** reprezintă index-ul liniei din **cache** în care trebuie să aducem date în caz de **CACHE_MISS** .

Algoritmul de load care trebuie implementat e următorul:

1. Calculăm tag-ul pentru adresa de la care vrem să obținem date
2. Iterăm prin tag-urile fiecărei linii. Dacă am găsit o linie care are asociat tag-ul calculat la pasul 1, calculăm offset-ul din cadrul liniei pentru octetul pe care vrem să-l copiem în registru și aducem acest octet din cache în registru. (Offset-ul e reprezentat de cei mai puțin semnificativ 3 biți din adresa octetului, cei pe care îi tăiem când calculăm tag-ul.)
3. Dacă nu am găsit o linie care are asociat tag-ul calculat la pasul 1, trebuie să aducem 8 octeți consecutivi din memorie într-o linie a cache-ului, unul din acești octeți fiind cel dorit (CACHE MISS). Linia pe care trebuie să o scriem la acest pas e cea cu index-ul **to_replace** . După ce a fost adusă linia în cache, registrul trebuie scris la fel ca la pasul precedent.

Exemplu

- Adresa de la care vrem să citim este, în binar: 01011010100010001010000011100**101**. Tag-ul este format din primii 29 de biți: **01011010100010001010000011100** . Offset-ul este format din ultimii 3 biți: **101** = 5. Dacă este găsit tag-ul în vectorul tags pe poziția **i** , se aduce direct din cache în registru valoarea **cache[i][offset] = cache[i][5]** .
- Dacă nu este găsit tag-ul în vector, se scriu cei 8 octeți cu adrese între 01011010100010001010000011100**000** și 01011010100010001010000011100**111** pe linia **cache[to_replace]**, după care se scrie în registru octetul **cache[to_replace]**

[offset]. De asemenea, **tags[to_replace]** devine 01011010100010001010000011100, astfel încât atunci când dorim să accesăm din nou o valoare din același interval de adrese, să o putem găsi direct în cache.

În cadrul exercițiului, va trebui să accesați valori de pe o anumită linie și coloană din matricea **cache**. Știți deja de la laborator cum să accesați date dintr-un vector. Accesarea valorilor din matricea alocată static este similară, deoarece chiar dacă noi o gândim ca fiind reprezentată pe linii și coloane, în realitate ea este reprezentată în memorie continuu, tot ce trebuie să facem e să găsim un mod de a transforma linia și coloana într-un singur index.

```
Pentru matricea alocată static:
1 2 3
4 5 6
7 8 9
Avem de fapt în memorie un șir continuu de forma:
1 2 3 4 5 6 7 8 9
```

Checker-ul nu verifică doar valoarea scrisă în final în variabila **reg**, ci și valorile scrise în matricea **cache** după efectuarea operației de load. Asta înseamnă că testele nu vor trece dacă încărcați mereu date din memorie, fără a interacționa cu cache-ul. De asemenea, vor fi teste care pică dacă aduceți mereu date din memorie în cache, și în caz de **CACHE HIT**, când nu ar trebui să faceți asta.

După cum puteți observa, în cadrul acestui exercițiu am realizat doar o simulare de lucru cu memoria cache, în realitate făcând doar transferuri de date în memoria RAM (atât registrul, cât și memoria cache sunt defapt stocate în memoria RAM). Transferurile de date care includ memoria cache nu sunt accesibile la nivel software, programele noastre având senzația că lucrează mereu cu RAM-ul (de aici și denumirea de memorie cache, adică memorie "ascunsă"). Pentru mai multe detalii despre memoria cache, puteți consulta acest articol: https://www.geeksforgeeks.org/cache-memory-in-computer-organization/ [https://www.geeksforgeeks.org/cache-memory-in-computer-organization/].

CUM VĂ SIMȚIȚI DUPĂ CE TERMINAȚI ULTIMUL TASK!

Precizări suplimentare

În schelet este inclus şi checker-ul, împreună cu testele folosite de acesta. Pentru a executa toate testele, se pot folosi comenzile `make run` sau `./checker`, după ce checker-ul a fost compilat, folosind `make`. Pentru a testa doar un subpunct, se poate folosi `./checker <n>` unde n este numărul subpunctului.

Formatul fişierelor de intrare pentru Reversed One Time Pad este:

```
len
plaintext
key
```

Formatul fişierelor de intrare pentru Ages este:

```
len
present
dates
```

Formatul fișierelor de intrare pentru Columnar Transposition Cipher este:

```
len_plaintext len_key
plaintext
key
```

Pentru cache, există 2 fișiere de intrare, care conțin datele ce se află în memoria RAM simulată. În checker, memoria RAM e simulată ca o matrice de dimensiune 16×16 pe primele 5 teste (care folosesc primul fișier de intrare), respectiv o matrice de dimensune 48×48 pentru următoarele 5 teste (care folosesc al doilea fișier de intrare). Acest acest lucru nu ar trebui să vă influențeze însă rezolvarea, deoarece veți primi direct adresa reală de memorie de la care trebuie să aduceți date. Formatul fișierelor de referință este următorul:

```
valoare registru după rulare load (reg)
cele 8 valori (octeți) care trebuie să se afle pe linia to_replace din cache după rulare load
```

Trimitere și notare

Temele vor trebui încărcate pe platforma vmchecker (în secțiunea IOCLA) și vor fi testate automat.

Arhiva încărcată va fi o arhivă `.zip` care trebuie să conțină:

- fișierele sursă ce conțin implementarea temei: `rotp.asm ages.asm columnar.asm cache.asm`
- `README` ce conține descrierea implementării

Punctajul final acordat pe o temă este compus din:

- punctajul obținut prin testarea automată de pe vmchecker - 90%
- README + coding style - 10%.

Coding style-ul constă în:

- prezența comentariilor în cod
- scrierea unui cod lizibil
- indentarea consecventă
- utilizarea unor nume sugestive pentru label-uri
- scrierea unor linii de cod/README de maxim 80-100 de caractere

Temele care nu trec de procesul de asamblare (build) nu vor fi luate în considerare.

Mașina virtuală folosită pentru testarea temelor de casă pe vmchecker este descrisă în secțiunea Mașini virtuale din pagina de resurse.

Vă reamintim să parcurgeți atât secțiunea de depunctări cât și regulamentul de realizare a temelor.

Resurse

Scheletul şi checker-ul sunt disponibile pe repository-ul de IOCLA [https://github.com/systems-cs-pub-ro/iocla/tree/master/teme/tema-2].

[1)] Funcția ceil(x) întoarce cea mai mică valoare întreaga, mai mare sau egală cu x

iocla/teme/tema-2.txt · Last modified: 2021/12/07 19:52 by ilinca_ioana.strutu