Tema 2 - The race is on

- Deadline: 16.05.2023
- Data publicării: 03.05.2022
- Responsabili:
    - Ilinca-Ioana Struțu
    - Rares Constantin
    - Serban Sorohan
    - Stefan Apostol
- Actualizări:
    - 03.05.2023 postare tema
    - 10.05.2023 clarificare detalii de implementare si de trimitere a temei
    - 12.05.2023 actualizare Makefile (task 3, task 4, Bonus), update enunt task 4

Enunț

Dupa cateva saptamani de pauza, inginerii echipelor de Formula 1 s-au intors la treaba. Fiind foarte multumiti de munca voastra anterioara, ei vor sa ii ajutati si acum.

Structura si detalii de implementare

Tema este formata din 4 exercitii independente si un exercitiu bonus. Fiecare task consta în implementarea unei sau mai multor functii in limbaj de asamblare. Implementarea se realizeaza in fisierele puse la dispozitie pentru fiecare exercitiu.

Parametrii functiilor sunt plasati in registre, in cadrul scheletului.

Scheletul include si macro-ul PRINTF32, folosit in laborator, pentru a va ajuta la depanarea problemelor. Tema finala nu trebuie sa faca afisari folosind PRINTF32, functii externe sau apeluri de sistem.

In tema finala este interzisa apelarea funcţiilor externe (ex. ne este acceptata implementarea rezolvarilor in C si apoi apelare functiilor in cadrul task-urilor). Este permisa utilizarea altor instructiuni decat cele prezentate la laborator/curs.

Task 1 - Simple cipher - 10p

Inginerul sef de la Ferrari trebuie sa le transmita celorlalti coechipieri mesaje criptate pentru a afla date despre masina. El vrea să folosească *simple cipher* pentru a transmite mesajele. Acest algoritm de criptare presupune shiftarea la dreapta în cadrul alfabetului a fiecărui caracter de un anumit număr de ori. De exemplu, textul *ANABANANA* se transformă

în *BOBCBOBOB* când pasul este 1. Astfel, o criptare cu pasul 26 nu modifică litera, întrucât alfabetul englez are 26 de caractere.
Pentru acest task va trebui să implementați în fișierul **simple.asm** funcția **simple()**, care criptează un string folosind metoda descrisă mai sus.
Antetul funcției este:

```
void simple(int n, char* plain, char* enc_string, int step)
```

Semnificația argumentelor este:

- **n** dimensiunea textului
- **plain** string-ul care trebuie criptat
- **enc_string** adresa la care se va scrie textul criptat
- **step** cu cât se shiftează fiecare caracter

Pentru ușurință se vor folosi doar majusculele alfabetului englez (A-Z), iar shiftarea se realizează strict în cadrul alfabetului englez cu o limită de 26 pentru step.

Task 2 - Processes (25p)

Intre timp, mecanicii de la RedBull incearca sa eficientizeze sistemul pe care il folosesc prin sortarea proceselor care ruleaza pe masina lor.

Prin intermediul acestui task se doreste aprofundarea lucrului cu structuri.

Se da structura simplificata a unui proces:

```
struct proc{

    short pid;

    char prio;

    short time;

};
```

Exercitiul 1

Pentru aceasta parte a task-ului aveti de implementat functia **sort_procs()** in fisierul **sort-procs.asm** care va simula sortarea tuturor proceselor active in momentul curent.
Pentru a intelege mai bine cum functioneaza un proces, vom explica mai jos ce inseamna fiecare field al structurii *proc*:

- Field-ul *pid* reprezinta id-ul unui proces care este prezent in sistem, fie el activ sau inactiv la momentul curent, acest id este unic fiecarui proces

- Field-ul *prio* reprezinta prioritatea pe care o are un proces atunci cand acesta ruleaza pe procesor. Fiecare proces are o astfel de prioritate, de la 1 la 5, 1 reprezentand prioritate maxima de rulare, iar 5 reprezentand prioritate minima de rulare. In functie de aceasta prioritate, procesele ajung sa ruleze mai devreme sau mai tarziu pe procesor
- Field-ul *time* reprezinta cuanta de timp acordata fiecarui proces in parte de a rula pe procesor. Desi veti intalni valori destul de mari in acest field pentru testare, in realitate nu exista cuante atat de mari de rulare, deoarece ar deveni unfair pentru restul proceselor sa astepte atat timp dupa un altul

Pentru a sorta procesele, stabilim urmatoarele reguli:

- Procesele trebuie sa apara in ordine crescatoare in functie de prioritate
- Pentru procesele cu aceeasi prioritate, acestea se vor ordona crescator in functie de cuanta de timp
- Pentru procese cu aceeasi prioritate si cu aceeasi cuanta de timp, acestea vor fi ordonate crescator dupa id

Sortarea se va face **in place**, adica vectorul *procs* prezentat mai jos va trebui, in urma apelului functiei, sa fie sortat. Antetul functiei este:

```
void sort_procs(struct proc *procs, int len);
```

Semnificatia argumentelor este:

- **procs** adresa de inceput a vectorului de procese - **len** numarul de procese aflate in sistem

Atentie! Nu puteti folosi functii externe pentru a sorta vectorul.

Exercitiul 2

In continuarea exercitiului 1, acum trebuie sa implementati functia **run_procs()** in fisierul **run_procs.asm** care va calcula intr-un mod simplificat timpul mediu de rulare pentru fiecare prioritate, adica va trebui sa calculati suma cuantelor de timp pentru o prioritate si apoi sa o impartiti la numarul de procese care au acea prioritate.
Pentru acest task va trebui sa declarati o structura **avg**,

```
    struct avg{

        short quo;

        short remain;

    };
```

unde:

- **quo** va stoca catul impartirii sumei cuantelor de timp la numarul de procese
- **remain** va stoca restul acestei impartiri

Va trebui sa puneti valorile obtinute in vectorul **avg_out** prezentat mai jos, pe prima pozitie aflandu-se rezultatul pentru prioritatea 1, iar pe ultima rezultatul pentru prioritatea 5. Antetul functiei este:

```
void run_procs(struct proc* procs, int len, struct avg *avg_out);
```

Semnificatia argumentelor este:

- **procs** adresa de inceput a vectorului de procese
- **len** numarul de procese aflate in sistem
- **avg_out** adresa de inceput a vectorului de structuri `avg`

Se garanteaza ca toate valorile raman in limitele tipurilor de date date in structura, adica **quo** si **remain** nu vor fi numere mai mari de 16 biti.

Pentru exercitiul 2 se va folosi acelasi vector folosit si la exercitiul 1. Daca anumite valori se modifica in urma exercitiului 1, atunci exercitiul 2 nu va putea fi rezolvat corect. Exercitiile sunt independente totusi, puteti alege sa nu faceti primul exercitiu, dar veti primi doar jumatate din punctajul aferent acestui task.

Task 3 - ENIGMA MACHINE (25p)

In timp ce inginerii de la Ferrari folosec simple_cipher, echipa Mercedes se intoarce in timp si alege sa foloseasca Enigma fiind convinsi ca rivalii lor de la Ferrari nu se vor descurca sa o decripteze.

Enigma machine este o masina de criptare complexa folosita de germani in al 2-lea Razboi Mondial pentru a encoda mesaje. Aceasta este alcatuita din mai multe componente:

1. Ordered List ItemPlugboard : o placa fizica care face swap intre anumite litere.
2. Reflector : o placa fizica care face swap intre litere 2 cate 2. De exemplu, daca A este legat la D, atunci D → (reflector) → A si A → (reflector) → D.
3. Rotor : o roata mecanica ce contine o permutare a celor 26 de litere ale alfabetului, si un notch care are ca rol rotirea rotii din stanga.

```
Exemplu : Pentru Rotor1 avem


A B C D E F G H I J K L M N O P Q R S T U V W X Y Z


E K M F L G D Q V Z N T O W Y H X U S P A I B R C J,
```

```
cu notch pe Q.
```

Vrem sa construim logica din spatele masinii enigma pentru a putea encripta si decripta mesaje.

Configuratia masinii este stocata intr-o matrice config[10][26] astfel :

- liniile 0 - 1 reprezinta configuratia primului rotor.
- liniile 2 - 3 reprezinta configuratia celui de-al doilea rotor.
- liniile 4 - 5 reprezinta configuratia celui de-al treilea rotor.
- liniile 6 - 7 reprezinta configuratia reflector-ului.
- liniile 8 - 9 reprezinta configuratia plugboard-ului.

1. Punerea rotorilor in pozitia initiala (5p):

Pentru rezolvarea acestui task aveti de implementat functi **rotate_x_positions** in fisierul **enigma.asm**. Antetul functiei este uramtorul:

```
void rotate_x_positions(int x, int rotor, char config[10][26], int forward)
```

unde:

- **x** - offset-ul noii pozitii`
- **rotor** - rotorul pe care se aplica rotatia (indexarea se face de la 0 !)`
- **config** - configuratia masinii pe care va trebui sa o modificati`
- **forward** - directia de rotatie` (daca forward = 0 atunci se shifteaza la stanga x pozitii, daca forward = 1 atunci se shifteaza la dreapta x pozitii).

Se va modifica matricea **config** conform parametrului rotor.

```
    Exemplu:

    Daca primele 2 linii din matricea config contin configuratia primului
rotor, de exemplu

    A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

    E K M F L G D Q V Z N T O W Y H X U S P A I B R C J

    si aplicam functia astfel:

    rotate_x_positions(3, 0, config, 0), atunci se modifica aceste 2 linii in

    D E F G H I J K L M N O P Q R S T U V W X Y Z A B C

    F L G D Q V Z N T O W Y H X U S P A I B R C J E K M
```

2. Codificarea mesajului (20p):

Pentru rezolvarea acestui task aveti de implementat functia **enigma** in fisierul **enigma.asm**. Antetul functiei este uramtorul:

```
void enigma(char *plain, char key[3], char notches[3], char config[10][26], char *enc)
```

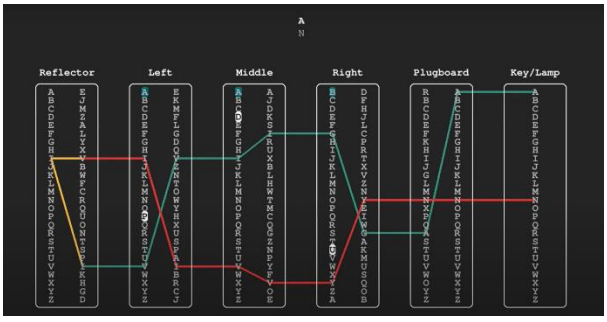
unde:

- **plain** - textul de criptat`
- **key** - pozitiile initiale ale rotorilor (key[i] = pozitia initiala a rotorului i)`
- **notches** - notch-urile initiale ale rotorilor (notches[i] = notch-ul initial a rotorului i)`
- **config** - configuratia masinii`
- **enc** - adresa la care va trebui sa scrieti textul criptat`

Atentie : inainte de citirea caracterului din plain, trebuie rotit al 3-lea rotor cu 1 pozitie (shiftare la stanga) si incrementata pozitia initiala a acestuia (alfabetul este circular deci daca incrementam Z acesta devine A). Daca pozitia curenta INAINTE DE ROTIRE a rotorului este egala cu notch-ul acestuia, atunci vom roti si rotorul din stanga acestuia cu 1 pozitie (acest lucru se va realiza pentru toti rotorii mai putin primul).
Exemplu 1 : key = "QWE", notches = "AAE" ⇒ key = "QXF"

Exemplu 2 : key = "QWE", notches = "AWE" ⇒ key = "RXF"

Exemplu de encriptare al unui caracter



Task 4 - Checkers (30p)

Pentru ca se plictisesc in garaj, mecanicii McLaren au decis ca tura asta sa nu mai joace UNO, ci sa joace dame.

Dorim sa simulam jocul de checkers (dame). Avem o matrice de 8×8 ce reprezinta suprafata de joc. Dandu-se pozitie unei dame pe suprafata de joc, dorim sa calculam noi pozitii pe care poate ajunge aceasta.

Pentru aceast task trebuie implementata functia **checkers** din fișierul **checkers.asm**. Antetul functiei este uramtorul:

```
void checkers(int x, int y, char table[8][8])
```

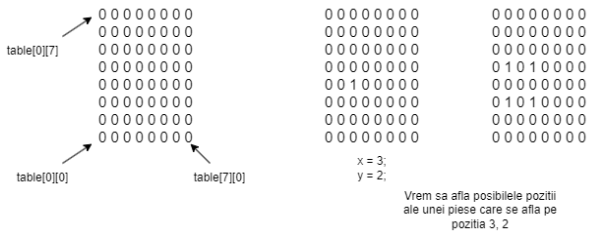
unde:

- x - linia pe care se afla piesa a carei pozitii vrem sa o calculam
- y - coloana pe care se afla piesa a carei pozitii vrem sa o calculam
- table - tabla de joc unde se vor pune pozitiile pe care poate ajunge piesa de joc

Nu vor exista coliziuni intre piesele de joc. Presupunem ca pe tabla de joc se afla o singura piesa.

Piesele nu pot sa iasa din suprafata de joc.

Exemplu



Bonus - Optimized checkers (20p)

Dorim sa optimizam reprezentarea suprafetei de joc de la Task 4 astfel incat sa ocupam mai putina memorie si calculele pozitiilor sa se faca mai rapid.

O posibila optimizare este reprezentata de notiunea de **Bitboard**. *Bitboard* reprezinta o structura de date binara in care fiecare bit reprezinta prezenta sau absenta unei piese pe o anumita pozitie a tablei de joc. De obicei, in C, pentru a reprezenta un bitboard, se foloseste o variabila de tip *unsigned long long*.
unsigned long lon este un tip de date ce contine 64 biti (8 octeti). O table de joc de dimensiunea 8×8 poate fi reprezentata intuitiv cu o singura variabila de acest tip, grupand, la nivel logic, cate 8 biti pentru fiecare linie din suprafata. Din pacate insa, noi nu avem acces la registri pe 64 biti, astfel incat trebuie folositi 2 registri pentru reprezentarea suprafetei de joc.
Pentru aceast task trebuie implementata functia **bonus** din fișierul **bonus.asm**. Antetul functiei este uramtorul:

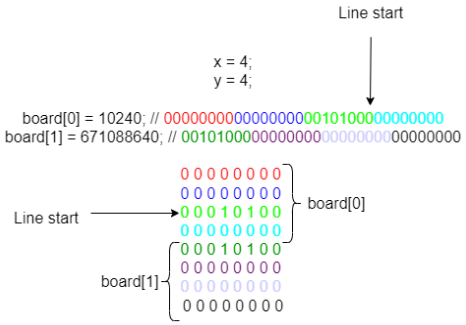
```
void bonus(int x, int y, int board[2])
```

unde:

- x - linia pe care se afla piesa a carei pozitii vrem sa o calculam

- y - coloana pe care se afla piesa a carei pozitii vrem sa o calculam
- board - doua numere intregi ce reprezinta suprafata de joc. Primul numar reprezinta partea superioara a suprafetei, pe cand al doilea numar reprezinta partea inferioara.

Exemplu



Precizări suplimentare

În schelet este inclus şi checker-ul, împreună cu testele folosite de acesta. Pentru a executa toate testele, se poate executa direct scriptul `checker.sh` din rădăcina temei:

```
python3 local_checker.py --all
```

Pentru a avea acces la output-ul functiilor, trebuie rulat checker-ul in cadrul folderului task-ului dorit:

```
make checher

./checker
```

Pentru a testa task-uri individual, folosiți:

```
python3 local_checker.py -t <număr_task>
```

Pentru a scrie rezolvarea unui task, intrați în directorul asociat task-ului respectiv și scrieți cod în fișierele în limbaj de asamblare indicate în enunț. **NU** modificați alte fișiere C, script-uri etc!

În cadrul unora din cele 5 task-uri, va trebui să accesați valori de pe o anumită linie și coloană dintr-o matrice. 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
```

Trimitere și notare

Temele vor trebui încărcate pe platforma Moodle, in cadrul assingment-ului Tema 2 și vor fi testate automat.
Folositi comanda:

```
python3 local_checker.py --zip
```

pentru a crea arhiva.

Punctajul final acordat pe o temă este compus din:

- punctajul obținut prin testarea automată - 90p
- coding style si comentarii- 10p
- bonus - TBD

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 (sau README) de maxim 80-100 de caractere

Pentru detalii despre coding style parcurgeti acest document: http://www.sourceformat.com/pdf/asm-coding-standard-brown.pdf

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

Arhivele care nu corespund structurii cerute vor fi depunctate cu 20 de puncte din nota finala.

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

FAQ

- **Q**: Este permisă utilizarea variabilelor globale?
    - **A**: Da
- **Q**: Este necesară parcurgerea laboratorului 8 pentru rezolvarea temei?
    - **A**: Nu, tema se poate rezolva doar cu materia din laboratoarelor 5,6,7, dar nu este depunctată utilizarea noțiunilor din laboratoarele următoare

Resurse

Scheletul şi checker-ul sunt disponibile pe repository-ul de IOCLA de pe GitLab.
Daca doriti sa folositi infrastructura de testare din cadrul GitLab, este nevoie sa va faceti un **fork privat** al repo-ului de tema.