პროგრამირების პარადიგმები - შუალედური - 2023/24

ინსტრუქციები - აუცილებლად დაიცავით

ნაშრომის დატოვება

დესკტოპზე უნდა დაგხვდეთ დირექტორია რომლის სახელიც თქვენი უნივერსიტეტის ელ-ფოსტის პრეფიქსია. ამ დირექტორიაში უნდა დააკოპიროთ თქვენი ნაშრომები საგამოცდოს დატოვებამდე.
გირჩევთ თავიდანვე მოცემულ დირექტორიაში დააკოპიროთ საკითხები და მანდ იმუშაოთ ამოხსნებზე, რომ ბოლოს კოპირება არ დაგავიწყდეთ.

ტერმინალის გამოყენება

კოდის დასაწერად გირჩევთ გამოიყენოთ VSCode რედაქტორი რომელიც შეგიძლიათ იპოვოთ დესკტოპზე. ამოცანაზე სამუშაოდ მისი ფაილების გასახსნელად გამოიყენეთ: **File > Open Folder** და აირჩიეთ დესკტოპზე გადმოწერილი საგამოცდო საკითხების დირექტორია. კოდის დასაკომპილირებლად და ტესტების გასაშვებად შეგიძლიათ გამოიყენოთ VSCode-ში ჩაშენებული ტერმინალი: **Terminal > New Terminal**

ასევე შეგიძლიათ გამოიყენოთ Windows-ის სტანდარტული ტერმინალი. გახსენით საკითხის დირექტორია, **compress** მაგალითად, და ცარიელ ადგილას ჯერ დააჭირეთ **Control + Shift + Mouse-Right-Click** და შემდეგ აირჩიეთ **Open Command Prompt**

კოდის დაკომპილირებიდან მის გაშვებამდე კარგად დააკვირდით კომპილაცია წარმატებით დასრულდა თუ არა. კომპილაცია თუ ვერ შესრულდა, ძველი დაკომპილირებული პროგრამა (a.exe მაგალითად) უცვლელი ანუ ძველი რჩება. თუ გინდათ რომ დარწმუნდეთ ახალ დაკომპილირებულ პროგრამას ამოწმებთ გირჩევთ კომპილაციამდე ძველი კომპილაციის შედეგი წაშალოთ, მაგალითად ტერმინალში გაუშვით del a.exe

საკითხებთან ერთად მოცემული ტესტები არის მხოლოდ სამაგალითო, რათა გაგიადვილდეთ ნაშრომების შემოწმება. საბოლოო შეფასების დათვლისას ნაშრომები გასწორდება ტესტების სხვა სიმრავლეზე.

Compress (20 ქულა)

თქვენი ამოცანაა იმპლემენტაცია გაუკეთოთ Compress ფუნქციას, რომელმაც კომპრესირება უნდა გაუკეთოს მოცემულ რიცხვების მიმდევრობას.

- მიმდევრობის თითოეული რიცხვი შედგება ზუსტად 4 **ბიტისგან** ანუ არის 0-დან 15-ის ჩათვლით
- შესაბამისად ერთ ბაიტში ინახება ორი რიცხვი, მაგალითად `0b01011101` ბაიტი აღწერს შემდეგ ორ რიცხვს 5 (`0101` - მარცხნიდან პირველი ოთხი ბიტი) და 13 (`1101` - ბოლო ოთხი ბიტი). `0bxxxx` ნოტაცია C-ში საშუალებას გაძლევთ რიცხვი ორობითში დაწეროთ.
- მოცემული რიცხვების რაოდენობა წინასწარ არ იცით, რიცხვების მიმდევრობა ბოლოვდება 4 ბიტიანი ნული რიცხვით. მაგალითად [`0b01011101` , `0b11110000`] შეიცავს სამ 4 ბიტიან რიცხს, ხოლო [`0b01011101` , `0b11110111` , `0b00000000`] შეიცავს ოთხ რიცხვს.

სრული მაგალითები:

- რიცხვების მიმდევრობა [1, 2, 3, 4] მეხსიერებაში წერია როგორც [`0b00010010` , `0b00110100` , `0b00000000`]
- რიცხვების მიმდევრობა [1, 2, 3, 4, 5] მეხსიერებაში წერია როგორც [`0b00010010` , `0b00110100` , `0b01010000`]

კომპრესირების შედეგად მიმდევრობით მდგომი ერთი და იგივე მნიშვნელობის რიცხვები ნაცვლდება წყვილით: რამდენჯერ მეორდება რიცხვი და გამეორებული რიცხვის მნიშვნელობა. **გარანტირებულია რომ ერთი და იგივე რიცხვი მიყოლებით 15-ზე მეტჯერ არ გვხვდება**. შესაბამისად რიცხვის გამეორების რაოდენობა და რიცხვის მნიშვნელობა შეიძლება 8 ბიტში (1 ბაიტში) ჩაიწეროს, სადაც პირველი ოთხი ბიტი აღნიშნავს განმეორების რაოდენობას, ხოლო მომდევნო ოთხი აღნიშნავს რიცხვის მნიშვნელობას.

მაგალითად თუ გვაქვს რიცხვების შემდეგი მიმდევრობა: [`0b00010001` , `0b01010101` , `0b01010000`]

- ეს ნიშნავს რომ ჯამში გვაქვს 5 რიცხვი [1, 1, 5, 5, 5]. პირველი ორი 4 ბიტიანი რიცხვია `0001` , მომდევნო სამი არის `0101` , ხოლო შემდეგ მოდის `0000` ანუ მიმდევრობის დაბოლოება.
- შესაბამისად ერთიანი მეორდება ორჯერ, ხოლო ხუთიანი - სამჯერ, ანუ ვიღებთ შემდეგი განმეორების წყვილებს: <2;1>, <3;5>
- <2;1> ერთ ბაიტში ჩაიწერება როგორც `0b00100001`
- <3;5> ერთ ბაიტში ჩაიწერება როგორც `0b00110101`
- შესაბამისად საწყისი რიცხვების მიმდევრობის კომპრესირებული ვერსიის შესანახად საჭიროა ორი ბაიტი.

კიდევ ერთი მაგალითი:

- [1, 1, 5, 5, 5, 1] ---> [<2;1>, <3;5>, <1;1>]
- [1, 1, 5, 5, 5, 1] ---> [<2-ჯერ;1-იანი>, <3-ჯერ;5-იანი>, <1-ხელ;1-იანი>]

ამ შემთხვევაში data და out მეხსიერებაში უნდა გამოიყურებოდეს ასე:

- [`0b00010001` , `0b01010101` , `0b01010001` , `0b00000000`] ---> | `0b00100001` , `0b00110101` , `0b00010001`]

int Compress(void* data, void* out):

- **data** მიმთითებელზე ჩაწერილია ბიტიანი რიცხვების მიმდევრობა
- უკან უნდა დააბრუნოთ თუ რამდენი ბაიტია საჭირო მიმდევრობის კომპრესირებული ვერსიის შესანახად
- ხოლო კომპრესირების შედეგი უნდა ჩაწეროთ **out** მისამართზე, სადაც უკვე წინასწარ გამოყოფილია საკმარისი მეხსიერება

გარანტირებულია, რომ:

- გადმოცემული void* მისამართი არანულოვანია.

ტესტირება

ნაშრომის დასაკომპილირებლად ტერმინალში გაუშვით: `gcc *.c` ხოლო ტესტებზე შესამოწმებლად: `a.exe`

---

struct_sum - **ასემბლერი** (40 **ქულა**)

თქვენი ამოცანაა struct_sum.c ფაილში არსებული C პროგრამირების ენაზე დაწერილი კოდი გადათარგმნოთ კურსზე გავლილ მანქანურ/ასემბლი კოდში. თარგმანი ჩაწერეთ struct_sum.s ფაილში, სადაც უკვე მონიშნულია თუ სად უნდა დაწეროთ თარგმნილი კოდი.

ფუნქციის გამოძახება მასზე მიმთითებლის გამოყენებით შეგიძლიათ გააკეთოთ `jalr` ინსტრუქციის გამოყენებით. მაგალითად ფუნქციის მისამართი ჯერ ჩაწერეთ `x10` რეგისტრში, ხოლო შემდეგ `jalr x10` გამოიძახებს ამ ფუნქციას.

თარგმნისას გაითვალისწინეთ რომ ქონვენშენების დაცვა საჭიროა როდესაც თქვენს კოდი გამოიძახებს სხვის ფუნქციას ან თქვენს კოდს გამოიძახებენ სხვა ფუნქციიდან. წინააღმდეგ შემთხვევაში ტესტები ჩაგეჭრებათ. ქონვენშენები რომლებიც უნდა დაიცვათ:

- ფუნქციის არგუმენტების სტეკზე ჩაწერა: ყველაზე მარცხენა არგუმენტი ჩაწერეთ ყველაზე ნაკლებ მისამართზე. მაგალითი: void fn(arg0, arg1) სტეკი: arg0: **sp**+0, arg1: **sp**+4... თქვენს დასაწერ ფუნქციაშიც არგუმენტები ასე დაგხვდებათ ჩაწერილი სტეკში.
- თქვენ მიერ დაწერილი ფუნქციებიდან დარეთარნდით, არ გამოიძახოთ exit().
- ფუნქციების return value შეინახეთ **x10** რეგისტრში. (ჩვენი ფუნქციის გამოძახება თუ დაგჭირდებათ ისიც **x10**-ში ჩაწერს return value-ს)
- ნებისმიერი ფუნქციის გამოძახებამ შეიძლება რეგისტრები გააფუჭოს ანუ ჩაწეროს სხვა მნიშვნელობები.
- სტრუქტურის field-ები მეხსიერებაში მიმდევრობითაა განლაგებული და გამოცდაზე შეგიძლიათ ჩათვალოთ, რომ მათ შორის padding-ები არ არის ანუ struct {short a, int b} არის 6 ზომის სტრუქტურა, სადაც a არის შენახული addr+0ზე და b addr+2ზე. (alignment-ზე რაც ვილაპარაკეთ დაივიწყეთ)
- არ აქვს მნიშვნელობა ლოკალური ცვლადებისთვის რა თანმიმდევრობით გამოყოფთ სტეკზე მეხსიერებას. ასევე არ აქვს მნიშვნელობა სტეკზე რა ადგილას შეინახავთ **ra**-ს და ა.შ.

თუ თქვენი ამოხსნა ტესტებს გადის, ყველაფერი სწორია.

ტესტირება

თქვენი ამოხსნის გასაშვებათ ტერმინალში შეასრულეთ შემდეგი ბრძანება: `java -jar ../venus.jar struct_sum.s`

Vector (40 **ქულა**)

თქვენი ამოცანაა იმპლემენტაცია გაუკეთოთ Vector ჯენერიკ სტრუქტურას შემდეგი ინტერფეისით:

- void VectorInit(Vector* v, int elem_size, FreeFn free_fn) - ინიციალიზაცია უნდა გაუკეთოს გადმოცემულ ვექტორს. გადმოგეცემათ შესანახი ელემენტების ზომა ბაიტებში და მეხსიერების გამათავისუფლებელი ფუნქცია (თუ ასეთი საჭიროა)
- void VectorDestroy(Vector* v) - უნდა გაათავისუფლოს ვექტორის და მასში შენახული ელემენტების მიერ გამოყენებული მეხსიერება
- void VectorAppend(Vector* v, void* elem) - ვექტორში ბოლო ელემენტად უნდა დაამატოს მოცემულ მისამართზე არსებული მნიშვნელობა
- void VectorInsert(Vector* v, int index, void* elem) - ვექტორში მოცემულ ინდექსზე უნდა ჩაამატოს გადმოცემულ მისამართზე არსებული მნიშვნელობა
- void VectorOverwrite(Vector* v, int index, void* elem) - ვექტორში მოცემულ ინდექსზე არსებულ ელემენტს უნდა გადააწერს გადმოცემულ მისამართზე არსებული მნიშვნელობა
- void* VectorGet(Vector* v, int index) - უნდა დააბრუნოს მოცემული ინდექსის ელემენტის მისამართი
- void VectorRemove(Vector* v, int index, void* elem) - ვექტორიდან უნდა ამოაგდოს მოცემული ინდექსის მქონე ელემენტი. თუ გადმოცემული**elem** მისამართი არანულოვანია, ელემენტის ამოგდებამდე მისი მნიშვნელობა უნდა დააკოპიროს ამ მისამართზე
- int VectorSize(Vector* v) - უნდა დააბრუნოს ვექტორში შენახული ელემენტების რაოდენობა

ვექტორის იმპლემენტაციის შემდეგ, მისი გამოყენებით იმპლემენტაცია უნდა გაუკეთოთ StudentVector სტრუქტურას რომლის ინტერფეისიც მორგებულია Student ობიექტების დამუშავებაზე. ფუნქციების უმრავლესობა უკვე იმპლემენტირებულია, თქვენ დაგჭირდებათ მხოლოდ ორი ახალი ფუნქციის შევსება:

- void StudentFree(void* elem) - უნდა გაათავისუფლოს სტუდენტის ობიექტის მიერ გამოყენებული დინამიურად გამოყოფილი მეხსიერება
- StudentVector StudentVectorCloneRange(StudentVector* v, int start, int end) - უკვე არსებული ვექტორის ელემენტები [start, end] დიაპაზონში (start-ის და end-ის ჩათვლით) უნდა დაკლონოს ახალ ვექტორში. ამ ფუნქციას ევალება ახალი StudentVector ობიექტის შექმნა, მისი ინიციალიზაცია და ძველი ვექტორიდან მოცემულ დიაპაზონში არსებული ელემენტების კლონირება და ახალ ვექტორში დამატება.

შეფასება

- VectorInit, VectorDestroy, VectorAppend, VectorGet ფუნქციების იმპლემენტაცია შეფასდება საკითხის შეფასების 40%-ით
- VectorInsert, VectorOverwrite, VectorRemove 40%-ით
- StudentVector -ის იმპლემენტაციაში 20%

ტესტზე თქვენი ნამუშევარი შეიძლება სწორ პასუხს აბრუნებდეს, მაგრამ მიუხედავად ამისა თუ თქვენი იმპლემენტაცია მეხსიერებას არასწორად იყენებს (მაგალითად free დაგავიწყდათ, ან უკვე დაბრუნებულ მეხსიერებას იყენებთ თავიდან) დაგაკლდებათ ტესტის შეფასების 15%.

ტესტირება

ნაშრომის დასაკომპილირებლად ტერმინალში გაუშვით: `gcc *.c` ხოლო ტესტებზე შესამოწმებლად: `a.exe`

<stdlib.h>

- void* malloc(size_t size); -- Allocates size bytes of uninitialized storage.
- void* realloc(void* ptr, size_t new_size); -- Reallocates the given area of memory. It must be previously allocated by malloc(), calloc() or realloc() and not yet freed with a call to free or realloc. Otherwise, the results are undefined.
- void free(void* ptr); -- Deallocates the space previously allocated by malloc(), calloc(), aligned_alloc, (since C11) or realloc().

<string.h>

- void* memcpy(void* dest, const void* src, size_t count); -- Copies count characters from the object pointed to by src to the object pointed to by dest. Both objects are interpreted as arrays of unsigned char.
- void* memmove(void* dest, const void* src, size_t count); -- Copies count characters from the object pointed to by src to the object pointed to by dest. Both objects are interpreted as arrays of unsigned char. The objects may overlap: copying takes place as if the characters were copied to a temporary character array and then the characters were copied from the array to dest. The behavior is undefined if access occurs beyond the end of the dest array. The behavior is undefined if either dest or src is a null pointer.
- char* strdup(const char* str1); -- Returns a pointer to a null-terminated byte string, which is a duplicate of the string pointed to by str1. The returned pointer must be passed to free to avoid a memory leak.
- char* strndup(const char* str, size_t size); -- Returns a pointer to a null-terminated byte string, which contains copies of at most size bytes from the string pointed to by str. If the null terminator is not encountered in the first size bytes, it is added to the duplicated string.
- int strcmp (const char* str1, const char* str2); -- Compares the C string str1 to the C string str2. This function starts comparing the first character of each string. If they are equal to each other, it continues with the following pairs until the characters differ or until a terminating null-character is reached. This function performs a binary comparison of the characters. For a function that takes into account locale-specific rules, see strcoll.
- int strncmp (const char* str1, const char* str2, size_t num); -- Compares up to num characters of the C string str1 to those of the C string str2. This function starts comparing the first character of each string. If they are equal to each other, it continues with the following pairs until the characters differ, until a terminating null-character is reached, or until num characters match in both strings, whichever happens first.