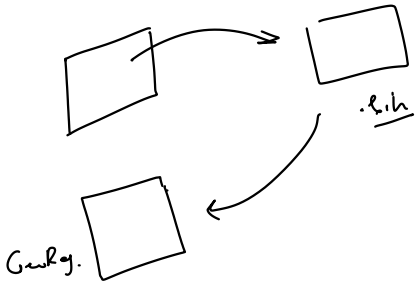
3. (10 poena)

Klasa GeoRegion, čiji je interfejs dat u nastavku, apstrahuje geografski region i implementirana je u potpunosti. Objekti ove klase zauzimaju mnogo prostora u memoriji, pa se mogu učitavati po potrebi, dinamički. Ovo učitavanje obavlja statička operacija GeoRegion::load, pri čemu se objekat identifikuje datim nazivom geografskog regiona (niz znakova). Ostale operacije ove klase vraćaju vrednosti nekih svojstava geografskog regiona.

Potrebno je u potpunosti implementirati klasu GeoRegionProxy čiji je intefejs dat u nastavku. Objekti ove klase služe kao posrednici (*proxy*) do objekata klase GeoRegion, pri čemu pružaju isti interfejs kao i „originali“, s tim da skrivaju detalje implementacije i tehniku dinamičkog učitavanja od svojih korisnika. Korisnici klase GeoRegionProxy vide njene objekte na sasvim uobičajen način, mogu ih kreirati datim konstruktorom i pozivati date operacije interfejsa, ne znajući da odgovarajuća struktura podataka možda nije učitana u memoriju.

```
class GeoRegion {
public:
  static GeoRegion* load (char* regionName);
  double getSurface ();
  double getHighestPeak ();
};

class GeoRegionProxy {
public:
  GeoRegionProxy (char* regionName);
  double getSurface ();
  double getHighestPeak ();
};
```




```
class GeoRegionProxy {
private:
    char * name;
    GeoRegion * region = nullptr;
public:
    GeoRegionProxy ( char * name) : name(name) {}
    double getSurface () {
        if ( region == nullptr ) {
            region = GeoRegion::load ( name);
        }
        return region -> getSurface();
    }
    double getHighestPeak () {
        if (region == nullptr) {
            region = GeoRegion:: load(name);
        }
        return region -> getHighestPeak();
    }
    ~ GeoRegionProxy () {
        if (region ≠ nullptr){
            delete region;
    }   }
```

stub

3. (10 poena)

U nekom sistemu postoje sledeći sistemski pozivi vezani za deljene biblioteke sa dinamičkim vezivanjem (DLL):

- `int mapDLL (const char* dllName)`: po potrebi učitava DLL sa zadatim imenom i mapira ga u virtuelni adresni prostor pozivajućeg procesa; u slučaju uspeha, vraća pozitivnu celobrojnu vrednost koja predstavlja identifikator datog DLL-a u kontekstu procesa; u slučaju greške, vraća negativnu vrednost;
- `int mapDLLSymbol (int dll, const char* symbolName)`: u DLL-u koji je prethodno mapiran pronalazi simbol sa zadatim imenom i vraća njegovu (virtuelnu) adresu; u slučaju neuspeha, vraća *null*.

U DLL-u „mydll.dll“ definisane su dve funkcije, `f1` i `f2`, čiji su potpisi dati dole. Simboli za njih kodovani su za povezivanje kao `f1@int@int*@int` i `f2@double@X*`, respektivno (ime f-je – povrtani tip – tipovi parametara). Korišćenjem datih sistemskih poziva realizovati „patrljke“ (*stub*) za ove dve funkcije koje program treba da poziva da bi pristupio njihovim implementacijama u DLL-u. U slučaju greške pozvati funkciju `handleError` koja ima isti potpis i ponašanje kao bibliotečna funkcija `printf`, samo što ispis šalje na standardni izlaz za greške i potom završava pozivajući proces.

```
int f1 (int*, int);
double f2 (X*);
```

```
int getDLL() {
    static int dll = 0;
    if (dll == 0) {
        dll = mapDLL("mydll.dll");
        if (dll < 0) {
            handleError("load dll failed\n");
        }
    }
    return dll;
}

int f1_stub(int* a1, int a2) {
    static int (f1*)(int*, int) = null;
    int dll = getDLL();
    if (f1 == null) {
        f1 = mapDLLSymbol(dll, "f1@int@int*@int");
        if (f1 == null) {
            handleError("map symbol f1 failed\n");
        }
    }
    return f1(a1, a2);
}
```

```
int f2_stub (X* a1) {
    static int (f2*)(int*, int) = null;
    int dll = getDLL();
    if (f2 ≠ null) {
        f2 = wrapDLLSymbol(dll, "f2@double@X*");
        if (f2* == null) {
            handleError("wrap symbol f2 failed\n");
        }
    }
    return f2(a1);
}
```

2. (10 poena)

Neki program koristi dinamičko učitavanje. Proces uvek zauzima kontinualan deo svog virtuelnog adresnog prostora određene potrebne veličine, počev od virtuelne adrese 0. Za svaki modul predviđen za dinamičko učitavanje prevodilac u glavnom modulu programa organizuje sledeću strukturu – deskriptor tog modula:

```
struct ModDesc {
  char* base;  // Base address of the module
  size_t size; // Size of the module in sizeof(char)
  const char* name;  // Name of the module's file
};
```

U polju `base` je početna virtuelna adresa modula, ukoliko je taj modul učitan; ako modul još nije učitan, ovo polje ima vrednost *null*. Polje `size` definiše veličinu modula, a polje `name` naziv fajla sa sadržajem modula.

a)(5) Na jeziku C napisati pomoćnu funkciju `load_module` koja treba da proširi alocirani deo virtuelnog adresnog prostora procesa za veličinu datog modula i učita taj modul u to proširenje, pod sledećim pretpostavkama:

```
void load_module (ModuleDesc* mod);
```

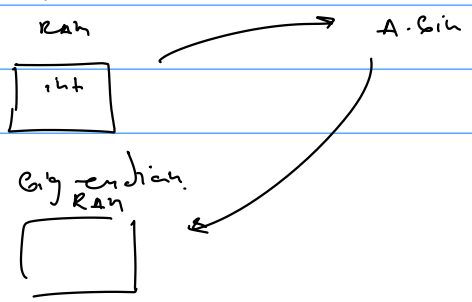
- globalna promenljiva programa `mem_size` tipa `size_t` sadrži trenutnu veličinu zauzetog (alociranog) dela virtuelnog adresnog prostora (u odnosu na početnu virtuelnu adresu 0); ovaj prostor treba proširiti i modul učitati u to proširenje;
- `mem_extend(size_t)`: sistemski poziv koji proširuje alocirani deo virtuelnog adresnog prostora za zadatu veličinu;
- `load(const char* filename, char* addr)`: sistemski poziv koji na zadatu adresu `addr` u virtuelnom adresnom prostoru procesa učitava sadržaj fajla sa zadatim imenom;
- ignorisati greške u sistemskim pozivima (ukoliko ne može da izvrši uslugu traženu sistemskim pozivom, sistem gasi pozivajući proces).

b)(5) Na asembleru nekog zamišljenog jednostavnog RISC procesora napisati kod koji koristi opisanu funkciju `load_module`, a koji prevodilac generiše za svaki poziv nekog potprograma `fun` koji se nalazi u nekom modulu koji se dinamički učitava, pod sledećim pretpostavkama:

- svi registri i pokazivači su 32-bitni, kao i sva polja u strukturi `ModDesc`; adresibilna jedinica je bajt;
- pre izvršavanja traženog koda, stvarni argumenti potprograma `fun` već su stavljeni na stek, a u registru R0 je adresa deskriptora modula (`ModDesc*`) u kome se nalazi potprogram `fun` (ovo prevodilac zna u toku prevođenja);
- pomeraj (relativna adresa u odnosu na početak modula) potprograma `fun` je poznat prevodiocu u toku prevođenja; ovaj pomeraj označiti simboličkom konstantom `fun`.

```
void load_module ( ModuleDesc * mod ) {
    mem_extend ( mod -> size );
    mod -> base = (char *)0 + mem_size;
    mem_size += mod -> size;
    load ( mod -> name, mod -> base);
}
```

```
fun_stub :  ld  R1 , (R0)
            jnz R1 , callfun
            push R0
            call load_modul
            pop  R0
            ld  R1 , (R0)
```



call fun : call #fun (Rw) int , < , > , * , /

ret

3. (10 poena)

U nekom fajlu zapisana su dva veoma velika celobrojna niza iste zadate veličine `arr_size`. Najpre je, počev od bajta 0, zapisano `arr_size` elemenata niza `a` tipa `int`, a odmah iza toga isto toliko elemenata niza `b`. Iza toga je u fajlu obezbeđen prostor za smeštanje istog tolikog niza `c`. Celobrojni elementi su veličine 4 bajta, a u fajlu su zapisani u istom formatu u kom se i smeštaju u operativnu memoriju (niži bajt na nižoj adresi, *little endian*).

Potrebno je realizovati funkciju `arr_add` koja treba da sabere ova dva niza `a` i `b` (element po element) i njihov rezultat zapiše na mesto obezbeđeno za niz `c` u datom fajlu. Ona se realizuje za neki skroman mikroračunar za vrlo malo RAM-a, tako da za smeštanje (delova) nizova u memoriju ukupno ne treba utrošiti više od 32 KB RAM-a. Na raspolaganju su funkcije `read_block` i `write_block` koje iz fajla učitavaju, odnosno u fajl upisuju (respektivno) `size` celih brojeva smeštenih na adresu datu pokazivačem `buffer`, počev od bajta sa rednim brojem `offset` u navedenom fajlu.

sizeof(int)

```
void read_block (size_t offset, size_t size, int* buffer);
void write_block (size_t offset, size_t size, const int* buffer);
void arr_add (size_t arr_size);
```

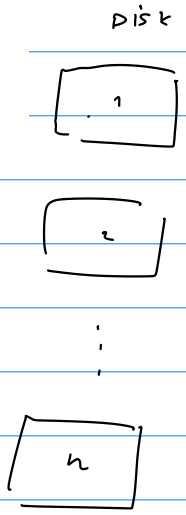
Rešenje:

$\frac{32\ KB}{4B} = 8 \cdot 2^{10}$ int-ova!

buff1 $4 \cdot 2^{10} = 2^{12}$

buff2 $4 \cdot 2^{10} = 2^{12}$.

↳ buff2 += buff1;



#define esize (1 << 12)

void arr_add() {

size_t offa = 0;

size_t offb = arr_size · sizeof(int); } između u bajtovima.

size_t offc = 2 · arr_size · sizeof(int);

size_t rem = arr_size; broj neobrađenih int-ova.

int buff1 [esize];

int buff2 [esize];

while (rem ≤ esize) {

read_block (offa, esize, buff1);

read_block (offb, esize, buff2);

```
        for (int i=0; i < esize; i++) {
            buff2[i] += buff1[i];
        }

        write_block (offc, bsize, buff2);
        offa += bsize * sizeof(int);
        offb += bsize * sizeof(int);
        offc += bsize * sizeof(int);
    }

    if (rem > 0) {
        read_block (offa, rem, buff1);
        read_block (offb, rem, buff2);
        for (int i=0; i < rem; i++) {
            buff2[i] += buff1[i];
        }
    }
}
```

2. (10 poena)

Neki sistem koristi preklope (*overlays*). Prevodilac i linker u generisanom kodu prevedenog programa koji koristi preklope statički alociraju i adekvatno inicijalizuju sledeće strukture podataka:

- za svaki modul (preklop, *overlay*) postoji sledeći deskriptor; moduli koji se preklapaju imaju istu početnu adresu:

```
struct OverlayDescr {
  const char* filename; // Naziv fajla u kome je binarni sadržaj preklopa
  void* addr; // Adresa u adresnom prostoru procesa na kojoj se nalazi
  bool isLoaded; // Da li je preklop trenutno učitan?
}
```

- tabela svih preklopa-modula:

```
const int numOfOverlays = ...; // Ukupan broj preklopa
OverlayDescr overlays[numOfOverlays];  // Tabela svih preklopa
```

- svakom potprogramu koji se nalazi u nekom preklopu prevodilac pridružuje jedan jedinstveni ceo broj (identifikator), koji predstavlja ulaz u tabelu tih potprograma; svaki ulaz u ovoj tabeli sadrži pokazivač na deskriptor preklopa u kome se nalazi taj potprogram:

```
const int numOfProcs = ...; // Ukupan broj potprograma
OverlayDescr* procedureMap[numOfProcs];  // Tabela potprograma
```

Za svaki poziv potprograma koji se nalazi u nekom preklopu, na uvek istoj i prevodiocu poznatoj adresi u adresnom prostoru procesa, npr. `proc(a,b,c)`, prevodilac generiše kod koji je ekvivalent sledećeg koda:

```
ensureOverlay (proc_id); // proc_id je identifikator za proc
proc(a,b,c); // standardan poziv na poznatoj adresi
```

Funkcija `ensureOverlay(int procID)` treba da obezbedi da je potprogram sa datim identifikatorom prisutan u memoriji, odnosno po potrebi učita njegov preklop. Potrebno je implementirati ovu funkciju, pri čemu je na raspolaganju sistemski poziv koji učitava binarni sadržaj iz fajla sa zadatim imenom na zadatu adresu u adresnom prostoru procesa:

```
void sys_loadBinary(const char* filename, void* address);
```

```
void ensureOverlay (int procID) {
    OverlayDescr * overlay = procedureMap [procID];
    if (overlay -> isLoaded) return;

    for (int i=0; i< numOfOverlays; i++) {
        if (overlays[i].addr == overlay->addr) {
            overlays[i].isLoaded = false;
        }
    }

    sys_loadBinary (overlay->filename, overlay->addr);
    overlay -> isLoaded = true;
}
```

2. (10 poena)

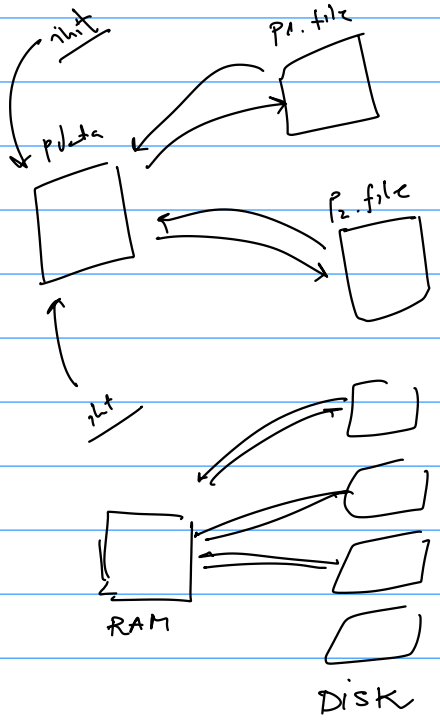
U nastavku je data implementacija jednog programa koji ciklično vrši neku obradu podeljenu u dve faze. Prva faza obrade koristi (čita i menja) podatke koji se samo koriste u toj fazi; svi ovakvi podaci grupisani su u jednu veliku strukturu podataka tipa `Phase1Data`; ali koristi i podatke koji su zajednički za obe faze, odnosno preko kojih obrade u ove dve faze razmenjuju informacije; ovi podaci grupisani su u strukturu tipa `CommonData`. Analogno radi i obrada u drugoj fazi.

Sa ciljem značajnog smanjenja potrebe za operativnom memorijom, potrebno je restrukturirati ovaj program tako da koristi preklop (*overlay*) u koji se smeštaju podaci prve, odnosno druge faze koji se ne koriste istovremeno. Za čuvanje izbačenog sadržaja treba kreirati dva privremena fajla u tekućem direktorijumu procesa. Za rad sa fajlovima na raspolaganju su sledeći sistemski pozivi; sve ove funkcije vraćaju 0 u slučaju uspeha, a negativnu vrednost u slučaju greške:

- `FILE fopen(char* filename)`: otvara fajl sa zadatim imenom za čitanje i upis; ako fajl ne postoji, kreira ga;
- `int fread(FILE file, int offset, int size, void* buffer)`: iz datog fajla, sa pozicije rednog broja bajta `offset` (numeracija počev od 0), čita niz bajtova dužine `size` u memoriju na lokaciju na koju ukazuje `buffer`; ukoliko se čitanjem prekorači granica sadržaja fajla, vraća grešku;
- `int fwrite(FILE file, int offset, int size, void* buffer)`: u dati fajl, na poziciju rednog broja bajta `offset` (numeracija počev od 0), upisuje niz bajtova dužine `size` sa lokacije na koju ukazuje `buffer`; ukoliko se upisom prekorači granica sadržaja fajla, sadržaj fajla se proširuje tako da primi sav upisani sadržaj;
- `int fclose(FILE file)`: zatvara dati fajl.

```
struct Phase1Data p1data;
struct Phase2Data p2data;
struct CommonData cdata;

void main () {
  init_cdata(&cdata);   // Initialize common data (cdata)
  init_p1data(&p1data); // Initialize data for phase 1 (p1data)
  init_p2data(&p2data); // Initialize data for phase 2 (p2data)

  do {
    process_phase_1(&p1data,&cdata);  // Perform phase 1 processing
    process_phase_2(&p2data,&cdata);  // Perform phase 2 processing
  } while (!cdata.completed);
}
```

```
struct CommonData cdata;

union pdata {
    struct Phase1Data p1;
    struct Phase2Data p2;
};

void main() {
    initit - cdata(&cdata);
    init p1 data (& pdata.p1);
    unload p1();
    init p2 data (& pdata.p2);
    unload p2();
```



```
while (1) {
    load_p1();
    process_phase_1(&pdata.p1, &cdata);
    unload_p1();
    load_p2();
    process_phase_2(&pdata.p2, &cdata);
    unload_p2();
}.

void load_p1() {                    "p1.file".
    FILE f1 = fopen("p1.file");
    fread(f1, 0, sizeof(Phase1Data), &pdata.p1);
    fclose(f1);
}

void unload_p1() {
    FILE f1 = fopen("p1.file");
    fwrite(f1, 0, sizeof(Phase1Data), &pdata.p1);
    fclose(f1);
}.

load_p2, unload_p2 ....
```