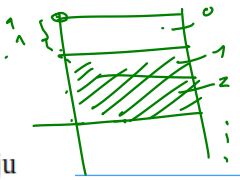
int mmap[MAX_BLK];



**2. (10 poena)**

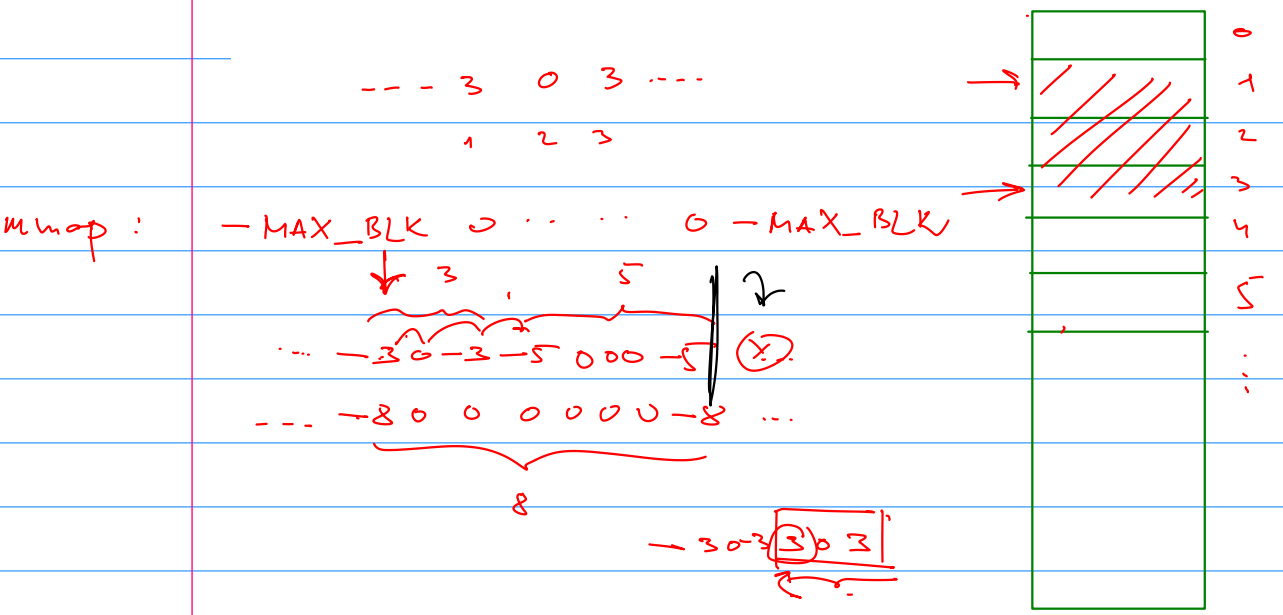
Neki sistem primenjuje kontinualnu alokaciju memorije za procese, s tim da tu memoriju alocira uvek u celim blokovima veličine `BLK_SIZE`, kako bi ublažio eksternu fragmentaciju. Alocira se uvek segment od najmanje dva bloka, pa i ispred prvog i iza poslednjeg alociranog segmenta ili više nema slobodnih blokova, ili postoji najmanje dva slobodna bloka. Evidencija slobodnih i zauzetih segmenata memorije vodi se u celobrojnom nizu `mmap` veličine `MAX_BLK`, tako što svaki element ovog niza odgovara po jednom bloku u prostoru za alokaciju procesa, redom. Za svaki alociran ili slobodan segment, u elementima niza `mmap` koji odgovaraju njegovom prvom i poslednjem bloku upisana je veličina tog segmenta izražena u blokovima, i to kao pozitivna vrednost ako je segment zauzet, odnosno kao negativna vrednost ako je segment slobodan.

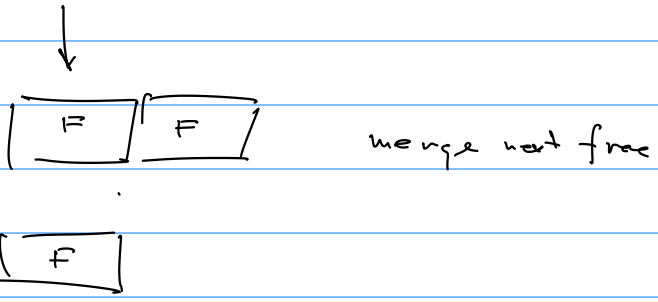
Implementirati najpre funkciju `mergeWithNextFree` koja slobodan segment koji počinje u datom bloku spaja sa segmentom iza njega, ukoliko je i taj segment slobodan, a onda i funkciju `freeSeg` koja oslobađa zauzet segment koji počinje u datom bloku, uz spajanje sa segmentom iza i ispred njega, ukoliko su slobodni.

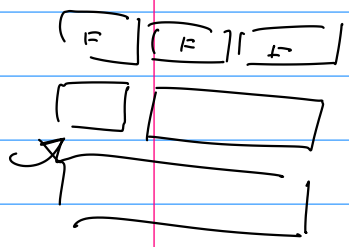
```
void mergeWithNextFree (int blk);
void freeSeg (int blk);
```

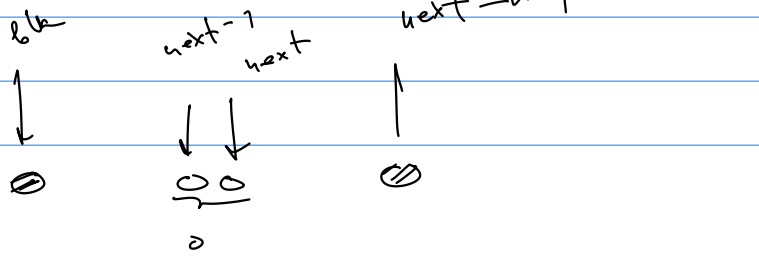
head free seg!

Rešenje:



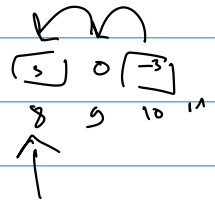





```
void mergeWithNextFree(int blk){
    int next = blk - mmap[blk];
    if (next < BLK_SIZE && mmap[next]<0){
        int new_size = mmap[next] + mmap[blk];
        mmap[blk] = new_size;
        mmap[next-1] = 0;
        mmap[next] = 0;
        mmap[next - mmap[next]-1] = new_size;
    }
}
```


```
void freeSeg(int blk){
    int blk_end = blk + mmap[blk]-1;
    mmap[blk] *= -1;
    mmap[blk_end] *= -1;

    mergeWithNextFree(blk);
    if (blk-1 >= 0 && mmap[blk-1] < 0){
        mergeWithNextFree(blk-1 + mmap[blk-1]);
    }
}
```




struct X{ int [100]; }

void *p

sizeof(char)

char* x = 0 ; x++ ; +

int * y = 0 ; y++ ; 4

X* z = 0. ; z++ ; 0+sizeof(X)=400;

malloc()

free.

int A[4], A+1

char *[4] A+1

# Septembar 2015. zadatak 2.

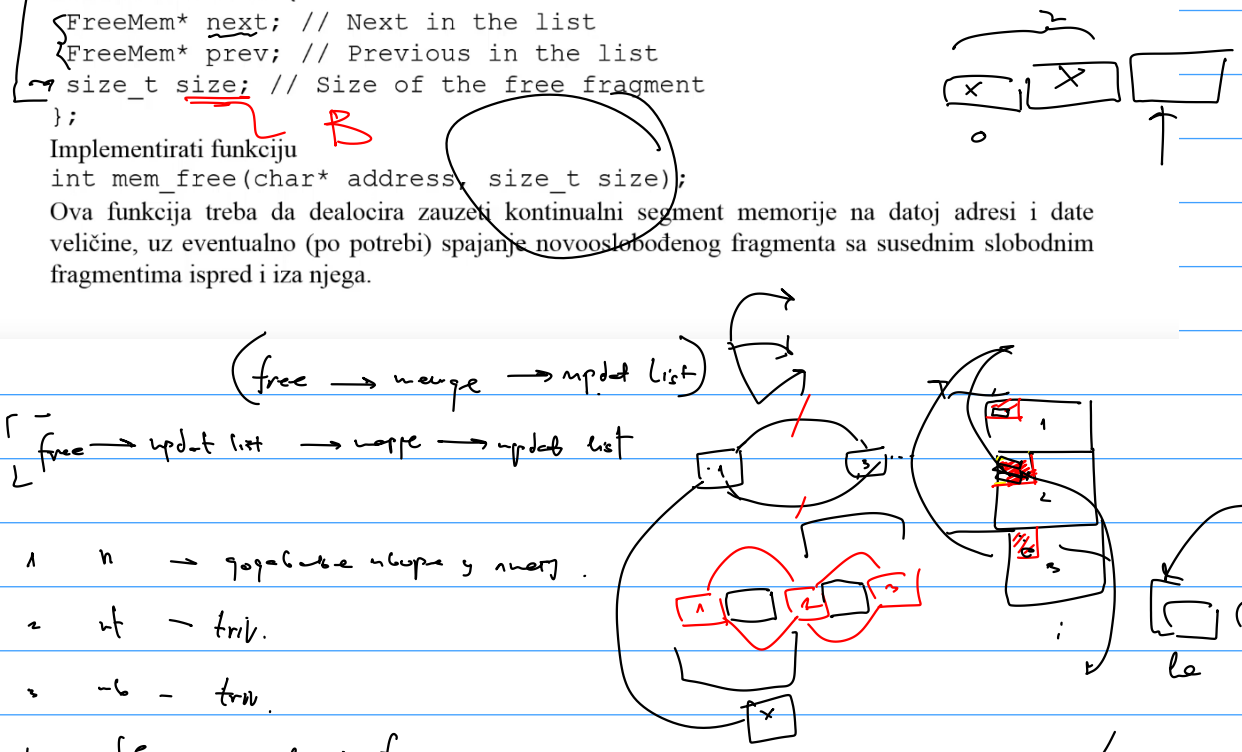
Neki sistem primenjuje kontinualnu alokaciju memorije. Slobodni fragmenti dvostruko su ulančani u listu na čiji prvi element ukazuje `fmem_head`. Fragmenti su ulančani u listu po rastućem redosledu svojih početnih adresa. Svaki slobodni fragment predstavljen je strukturom `FreeMem` koja je smeštena na sam početak tog slobodnog fragmenta:

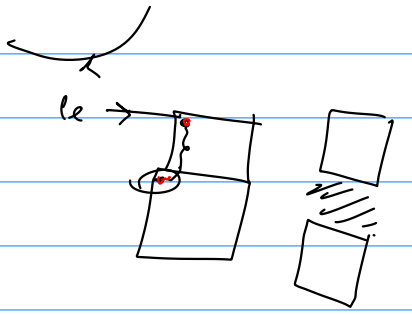
```
struct FreeMem {
  FreeMem* next; // Next in the list
  FreeMem* prev; // Previous in the list
  size_t size; // Size of the free fragment
};
```

Implementirati funkciju

```
int mem_free(char* address, size_t size);
```

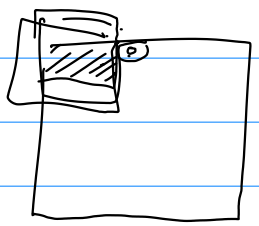
Ova funkcija treba da dealocira zauzeti kontinualni segment memorije na datoj adresi i date veličine, uz eventualno (po potrebi) spajanje novooslobođenog fragmenta sa susednim slobodnim fragmentima ispred i iza njega.



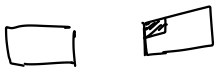




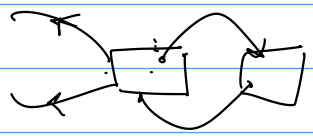
```
void MergeAfter(FreeMem *le) {
    FreeMem *next = le->next;
    if (next != null && ((char*)le + le->size) == (char*)next) {
        le->size += next->size
        le->next = next->next.
        if (le->next->next != null) {
            le->next->next->prev = le;
        }
    }
}
```



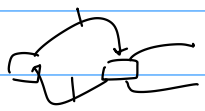
fmem_head — NULL

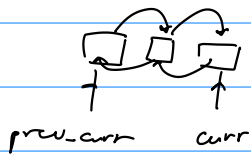


```
int mem_free(char* address, sizet size){
    FreeMem * le = (FreeMem*) address;
    le -> next = NULL;
    le -> prev = NULL;
    le -> size = size;

    if (fmem_head == NULL){
        fmem_head = address;
    } else {
        FreeMem * prev_curr = null;
        FreeMem * curr = fmem_head;
        while (curr != NULL && curr < address) {
            prev_curr = curr;
            curr = curr -> next;
        }
        if (curr == NULL){
            FreeMem * tail = prev_curr;
            tail -> next = address;
            le address -> prev = tail;
        } else {
            if (prev_curr == null) {
                le address -> next = curr;
                curr -> prev = address;
                fmem_head = address;
            } else {
                prev_curr -> next = address;
                le address -> next = curr;
                curr -> prev = address;
                le. address -> prev = prev_curr;
            }
        }
    }
```

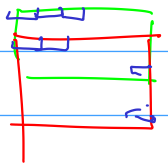


curr == fmem_head





```
mergeAfter(le)
if (le->prev != null) {
    mergeAfter(le->prev);
}
}
```



Process Control Block

**2. (10 poena)**

U nekom sistemu koristi se kontinualna alokacija memorije. U PCB postoje sledeća polja:

- `char* mem_base_addr`: početna adresa u memoriji na koju je smešten proces;
- `size_t mem_size`: veličina dela memorije koju zauzima proces; → B
- `PCB* mem_next`: pokazivač na sledeći u listi procesa; PCB procesa su ulančani u ovu listu po redosledu njihovog smeštanja u memoriji (po rastućem redosledu `mem_base_addr`); glava ove liste je globalna promenljiva `proc_mem_head`.

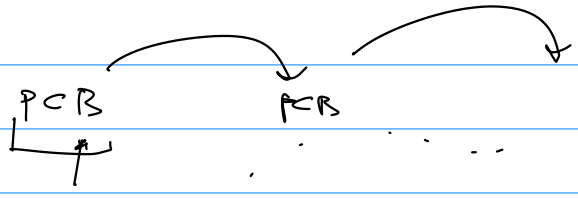
Na početak dela operativne memorije koja se koristi za smeštanje procesa ukazuje pokazivač `user_proc_mem_start`, a na njegov kraj (na poslednji znak) pokazivač `user_proc_mem_end`; oba su tipa `char*`.

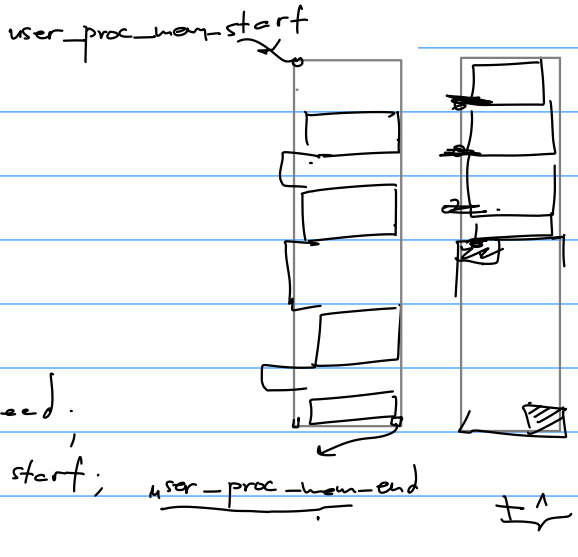
Fragmenti slobodne memorije ulančani su u jednostruko ulančanu listu, pri čemu se svaki element te liste, tipa `FreeSegment`, smešta na sam početak slobodnog fragmenta. Glava ove liste je u globalnoj promenljivoj `mem_free_head`. Struktura `FreeSegment` izgleda ovako:

```
struct FreeSegment {
  FreeSegment* next;  // Next free segment in the list
  size_t size; // Total size of the free segment
};
```

Napisati proceduru `mem_compact()` koja vrši kompakciju slobodnog prostora relokacijom procesa.

Rešenje:



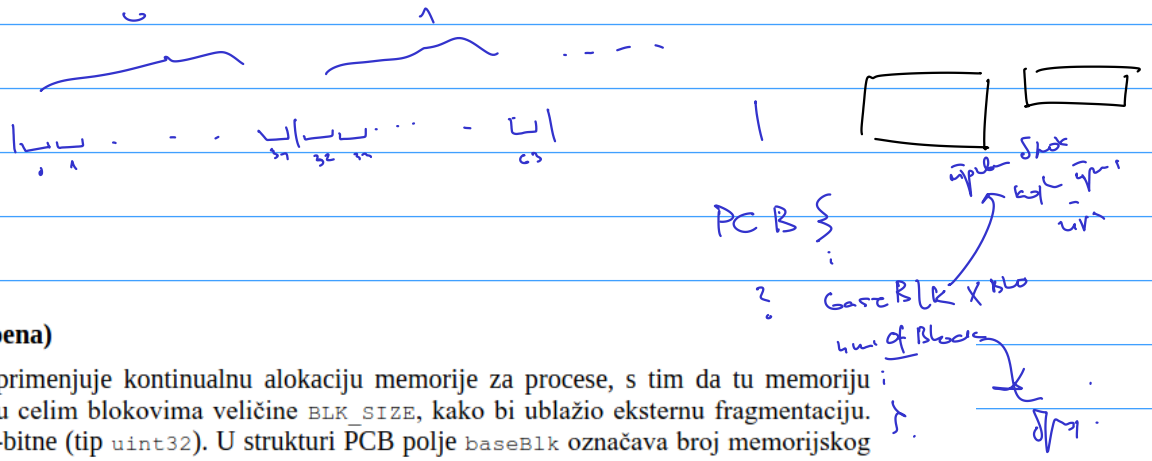

```
void mem_compact() {
    PCB * curr = proc_mem_head;
    char * start = user_proc_mem_start;
    while (curr != null) {
        for (int i=0; i < curr->size; i++) {
            start[i] = curr->mem_base_addr[i];
        }
        curr->mem_base_addr = start;
        start += curr->size;
        curr = curr->next;
    }
```

free_mem_head = (FreeSegment *) start;

free_mem_head → next = null;

free_mem_head → size = (user_proc_mem_end − start) + 1;

}



**2. (10 poena)**

Neki sistem primenjuje kontinualnu alokaciju memorije za procese, s tim da tu memoriju alocira uvek u celim blokovima veličine `BLK_SIZE`, kako bi ublažio eksternu fragmentaciju. Adrese su 32-bitne (tip `uint32`). U strukturi PCB polje `baseBlk` označava broj memorijskog bloka koji je prvi alociran za proces (proces zauzima uvek memoriju počev od pomeraja 0 tog bloka), a polje `numOfBlocks` označava broj blokova memorije koje proces zauzima; proces uvek zauzima cele alocirane blokove i poravnat je na početak bloka. Evidenciju o slobodnim blokovima sistem vodi u bit-vektoru `freeMemBlks`; svaki bit ovog vektora predstavlja jedan blok memorije (redom od nižih prema višim elementima niza i redom od najnižeg do najvišeg bita svakog 32-bitnog elementa niza, blokovi su numerisani počev od 0), a vrednost 1 označava da je blok slobodan.

a)(2) Implementirati funkciju `getMemCtxt` koja treba da izračuna baznu adresu (i upiše je u parametar `base`) i najvišu dozvoljenu virtuelnu adresu (i upiše je u parametar `limit`) za dati proces. Ovu funkciju sistem poziva prilikom promene konteksta da bi izračunate vrednosti upisao u odgovarajuće registre procesora kada datom procesu dodeljuje procesor.

b)(3) Implementirati funkciju `isMemBlkFree` koja za dati redni broj memorijskog bloka vraća informaciju o tome da li je dati blok slobodan ili ne, kao i funkciju `allocMemBlk` koja memorijski blok sa zadatim brojem označava zauzetim.

c)(5) Implementirati funkciju `expand` koju sistem poziva kada tekući proces izazove izuzetak zbog prekoračenja najviše dozvoljene virtuelne adrese. Ova funkcija treba da pokuša da dodeli još jedan dodatan memorijski blok datom procesu, ako takav postoji iza prostora koji proces već zauzima, proširujući tako memorijski prostor procesa; u tom slučaju ova funkcija treba da vrati 0. U suprotnom, ova funkcija treba da vrati -1. Najviši blok fizičke memorije je uvek zauzet od strane jezgra operativnog sistema i nikad se ne dodeluje procesima.

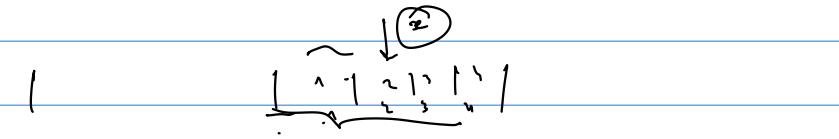
```
extern uint32 freeMemBlks[];
extern size_t BLK_SIZE;

void getMemCtxt (PCB* pcb, uint32& base, uint32& limit);

bool isMemBlkFree (size_t num);
void allocMemBlk (size_t num);

int expand (PCB* pcb);
```

void getMemCtxt (PCB *pcb, uint32& base, uint32& limit) {

base = pcb → baseBlk · BLK_SIZE;

limit = base + pcb → numOfBlks · BLK_SIZE − 1;

}



uint32 freeMemBlk[...]

32.

```
bool isMemBlkFree(size_t num){
    int idx = num / 32;
    int off = num % 32;
    return (freeMemBlk[idx] & ((uint32)1 << off)) > 0
}

int expand(PCB *pcb){
    if(isMemBlkFree(pcb->base + pcb->numOfBlocks)){
        allocMemBlk(pcb->base + pcb->numOfBlocks);
        pcb->numOfBlocks += 1
        return 0;
    } else {
        return -1;
    }
}

void allocMemBlk(size_t num){
    int idx = num / 32;
    int off = num % 32;
    freeMemBlk[idx] = ~((uint32)1 << off);
}
```