Elektrotehnički fakultet u Beogradu
Katedra za računarsku tehniku i informatiku

Predmet: Operativni sistemi 1
Nastavnik: prof. dr Dragan Milićev
Odsek: Softversko inženjerstvo, Računarska tehnika i informatika
Kolokvijum: Integralni, avgust 2021.
Datum: 5. 9. 2021.

Integralni kolokvijum iz Operativnih sistema 1

Kandidat: _____

Broj indeksa: _____ *E-mail:*_____

Kolokvijum traje 2 sata. Dozvoljeno je korišćenje literature.

Zadatak 1	_____/10	*Zadatak 3*	_____/10
Zadatak 2	_____/10	*Zadatak 4*	_____/10

Ukupno: _____/40

Napomena: Ukoliko u zadatku nešto nije dovoljno precizno definisano, student treba da uvede razumnu pretpostavku, da je uokviri (da bi se lakše prepoznala prilikom ocenjivanja) i da nastavi da izgrađuje preostali deo svog odgovora na temeljima uvedene pretpostavke. Ocenjivanje unutar potpitanja je po sistemu "sve ili ništa", odnosno nema parcijalnih poena. Kod pitanja koja imaju ponuđene odgovore treba **samo zaokružiti** jedan odgovor. Na ostala pitanja odgovarati **čitko, kratko i precizno**.

1. (10 poena)

Implementirati bibliotečnu funkciju `malloc` za dinamičku alokaciju memorijskog bloka veličine `size` (sve deklaracije date su dole). Ako nema dovoljno slobodne memorije, ova funkcija vraća `nullptr`.

Evidencija slobodnih blokova u virtuelnom adresnom prostoru procesa koju vodi ovaj alokator organizovana je kao neuređena jednostruko ulančana lista sa glavom u `freeMemHead`. Svaki element liste je jedan slobodan deo memorije koji na početku ima zaglavlje tipa `BlockHeader` u kome je pokazivač na sledeći takav zapis, kao i veličina slobodnog bloka iza ovog zaglavlja (neto, bez veličine zaglavlja). Funkcija `malloc` najpre traži slobodan blok dovoljne veličine u ovoj svojoj evidenciji slobodnih blokova algoritmom *first fit*. Ako takav blok ne nađe, ona alocira nov logički segment veličine `BLOCK_SIZE` ili tražene veličine `size` (šta god je veće) u virtuelnom adresnom prostoru sistemskim pozivom `mmap` i dodaje ga u svoju evidenciju. Ukoliko bi nakon alokacije ostao slobodan fragment veličine manje od `MIN_BLOCK_SIZE`, alocira se ceo slobodan blok (drugim rečima, ne ostavlja se slobodan fragment manji od `MIN_BLOCK_SIZE`). Da bi funkcija `free` za oslobađanje memorijskog bloka znala njegovu veličinu, uz alociran blok ostaje zaglavlje, s tim da funkcija `malloc` vraća pokazivač na slobodan blok ispred koga je zaglavlje, a ne na zaglavlje.

```
void* malloc (size_t size);

struct BlockHeader {
  BlockHeader* next;
  size_t size;
};
static BlockHeader* freeMemHead;

const size_t BLOCK_SIZE = ..., MIN_BLOCK_SIZE = ...;

void* mmap (void* addr, size_t size, int protection);
const int PROT_RD = ..., PROT_WR = ..., PROT_RW = PROT_RD | PROT_WR;
```

Rešenje:

2. (10 poena)

Školsko jezgro proširuje se nestatičkom funkcijom `Thread::join()` koju sme da pozove samo roditeljska nit date niti da bi sačekala da se data nit-dete završi; ukoliko ovu operaciju pozove neka druga nit koja nije roditelj date niti, ova funkcija vraća grešku (-1). Kada jezgro pravi novu nit, poziva funkciju `Thread::wrapper` koja obavlja sve potrebne radnje pre poziva funkcije `Thread::run` (u kontekstu napravljene niti) i nakon povratka iz nje:

```
void Thread::wrapper (Thread* toRun) {
  ...
  unlock();
  toRun->run();
  lock();
  ...
}
```

Precizno navesti sve izmene i dopune koje je potrebno napraviti u klasi `Thread` i implementirati operaciju `Thread::join`. Uzeti u obzir to da nit-roditelj može pozvati `join` više puta, pri čemu se samo pri prvom pozivu eventualno može zaustaviti dok nit-dete ne završi, svi pozivi nakon toga su neblokirajući. Problem gašenja niti i brisanja objekta klase `Thread`, kao i sinhronizacije potrebne za to ne treba rešavati u ovom zadatku.

Rešenje:

3. (10 poena)

U računaru postoji ulazni uređaj koji prihvata pakete sa neke komunikacione veze. Paket je veličine `PACKET_SIZE` (u `sizoef(char)`). Kada paket stigne u interni hardverski bafer uređaja, uređaj generiše prekid na koji je vezana rutina `packetArrived` kernela. Kernel treba da prebaci paket u svoj interni memorijski kružni bafer `packetBuffer` koji ima mesta za `BUFFER_SIZE` paketa. Ovo prebacivanje kernel obavlja DMA prenosom. Ostatak kernela te pakete iz kružnog bafera kernela isporučuje dalje kome treba (procesima). Kada prebaci paket u svoj kružni bafer, kernel treba to da signalizira uređaju upisom konstante `IO_COMPLETE` u upravljački registar uređaja, kako bi ovaj znao da može da prihvati naredni paket u svoj interni bafer. Ukoliko je kružni bafer kernela pun ili pri DMA prenosu dođe do greške, kernel treba uređaju da javi da paket mora da se odbaci upisom konstante `IO_REJECT` u upravljački registar uređaja. Date su deklaracije pokazivača preko kojih se može pristupiti registrima DMA kontrolera i kontrolera ulaznog uređaja, kao i deklaracije za kružni bafer:

```
typedef unsigned int REG;
REG* dmaCtrl =...;    // DMA control register
REG* dmaStatus =...; // DMA status register
REG* dmaAddress =...;    // DMA block address register
REG* dmaCount =...;    // DMA block size register
REG* ioCtrl =...;   // Input device control register

extern const size_t PACKET_SIZE, BUFFER_SIZE;
extern char packetBuffer[BUFFER_SIZE*PACKET_SIZE];
extern char *pbHead, *pbTail;

interrupt void packetArrived ();
interrupt void dmaComplete ();
```

Greška u DMA prenosu signalizira se postavljanjem bita u statusnom registru; ovaj bit maskira se konstantom `DMA_ERROR`. DMA prenos pokreće se upisom konstante `DMA_START` u upravljački registar DMA kontrolera. Kada DMA kontroler završi prenos, generiše prekid na koji je vezana rutina `dmaComplete` kernela. Po završetku prenosa ne treba ništa upisivati u upravljački registar DMA kontrolera. Glava `pbHead` ukazuje na prvi znak paketa u kružnom baferu koji je na redu za prenos u ostatak kernela, a rep `pbTail` ukazuje na prvi znak prvog slobodnog mesta za prijem novog paketa u kružni bafer. Tokom izvršavanja prekidnih rutina drugi prekidi su maskirani i nema uporednog izvršavanja ostalog koda kernela.

Realizovati prekidne rutine `packetArrived` i `dmaComplete`.

Rešenje:

4. (10 poena)

U nekom FAT fajl sistemu FCB-ovi su smešteni na jednom delu particije, a sadržaj fajlova na drugom, disjunktnom delu particije. Za taj fajl sistem pravi se sistemski program koji radi kompakciju prostora na disku tako što premešta blokove sa sadržajem fajla (ne i one sa FCB-om) tako da budu susedni na particiji. FAT tabela je veličine `FAT_SIZE` i njeni ulazi predstavljaju sve blokove na particiji. Ulaz FAT-a sadrži broj sledećeg bloka u ulančanoj listi blokova sa sadržajem fajla, odnosno u ulančanoj listi slobodnih blokova čija je glava u globalnoj promenljivoj `freeBlkHead`; nula u ulazu označava kraj liste blokova sa sadržajem, a negativna vrednost u ulazu označava slobodan blok (sledeći u lancu slobodnih je dat suprotnom vrednošću te negativne, -1 označava kraj liste). FCB svakog fajla smešten je u jedan blok na prvom delu particije, a ulaz FAT-a koji odgovara tom bloku sadrži broj prvog bloka sa sadržajem tog fajla. FAT je cela keširana u memoriji.

Implementirati funkciju `compact` koja za dati fajl radi kompakciju premeštanjem blokova sa sadržajem tog fajla tako da budu smešteni redom počev od bloka sa brojem `startBlk`. Parametar `fcbBlk` određuje broj bloka u kom je FCB datog fajla. Prostor počev od bloka broj `startBlk` nije oslobođen (u njemu se možda nalaze blokovi sa sadržajem fajlova), ali do kraja particije svakako ima dovoljno blokova za smeštanje sadržaja datog fajla. Na raspolaganju su operacije za čitanje i upis bloka sa datim brojem za dati bafer u memoriji veličine `BLK_SIZE` znakova. Ignorisati greške.

```
int FAT[FAT_SIZE];
void readBlock (int blk, char* buffer);
void writeBlock(int blk, const char* buffer);
void compactFile (int fcbBlk, int startBlk);
```

Rešenje: