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, oktobar 2020.
Datum: 30. 9. 2020.

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)

U sistemu Windows postoji sistemski poziv

```
int spawnvp (int mode, char *path, const char **argv);
```

koji radi isto što i kombinacija poziva *fork* i *exec* na sistemima nalik sistemu Unix: kreira proces dete pozivajućeg procesa nad programom u izvršivom fajlu zadatom putanjom u argumentu `path`. Parametar `argv` predstavlja niz pokazivača na nizove znakova (*null-terminated strings*) koji predstavljaju argumenate poziva programa koji treba izvršiti. Po konvenciji, prvi argument pokazuje na naziv fajla samog programa koji se izvršava. Ovaj niz pokazivača mora da se završi elementom sa vrednošću `NULL`. Ovaj sistemski poziv vraća 0 u slučaju uspeha, a vrednost manju od 0 u slučaju greške. Parametar `mode` definiše modalitet kreiranja procesa deteta sledećim simboličkim konstantama:

P_OVERLAY *Overlays the parent process with child, which destroys the parent. This has the same effect as the exec functions.*

P_WAIT *Suspends the parent process until the child process has finished executing (synchronous spawn).*

P_NOWAIT *Continues to execute the parent process concurrently with child process (asynchronous spawn).*

P_DETACH *The child is run in the background without access to the console (asynchronous spawn).*

Korišćenjem ovog sistemskog poziva realizovati sledeću funkciju:

```
int multispawn (int number, const char* path, const char* args[]);
```

Ova funkcija treba da kreira zadati broj (`number`) procesa dece, i svaki od tih procesa dece treba da izvršava isti program zadat u fajlu sa stazom `path`, sa samo po jednim argumentom. Argumenti tih procesa dece zadati su redom u prvih `number` elemenata niza `args`. Ova funkcija treba da vrati broj uspešno kreiranih procesa dece i da odmah vrati kontrolu pozivaocu, ne čekajući da se ti procesi deca završe.

Rešenje:

2. (10 poena)

Neki sistem koristi segmentnu organizaciju memorije. Slobodne fragmente operativne memorije sistem opisuje strukturama tipa `FreeSegDesc` za svaki proces. Kako bi pretraga pri alokaciji memorije bila efikasnija, ovi desktiptori organizovani su u uređeno binarno stablo, tako da su u levom podstablu svakog čvora deskriptori slobodnih fragmenata koji su manji ili jednaki, a desno onih koji su veći od fragmenta datog čvora. U strukturi `FreeSegDesc` polja `left` i `right` ukazuju na koren levog odnosno desnog podstabla, polje `addr` sadrži početnu adresu fragmenta, a polje `sz` sadrži veličinu fragmenta. Tip `size_t` je neoznačen celobrojni tip dovoljno velik da predstavi veličinu adresnog prostora.

```
struct FreeSegDesc {
  FreeSegDesc *left, *right;
  size_t sz;
  void* addr;
  ...
};
```

Implementirati operaciju koja pretragom u stablu na čiji koren ukazuje prvi parametar pronalazi i vraća deskriptor slobodnog fragmenta u koji se može alocirati segment zadate veličine, a *null* ako takvog nema, i to radi primenom *best fit* algoritma alokacije. Ova operacija ne treba da ažurira ni stablo ni pronađeni deskriptor, već samo da pronađe i vrati pokazivač na odgovarajući deskriptor.

```
SegDesc* findSegDesc (SegDesc* root, size_t size);
```

Rešenje:

3. (10 poena)

Date su deklaracije pokazivača preko kojih se može pristupiti registrima jednog DMA kontrolera:

```
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
```

U upravljačkom registru najniži bit je bit *Start* kojim se pokreće prenos jednog bloka preko DMA, a u statusnom registru najniži bit je bit završetka prenosa (*TransferComplete*), a bit do njega bit greške (*Error*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`).

Zahtevi za ulaznim operacijama na nekom uređaju sa kog se prenos blokova vrši preko ovog DMA kontrolera vezani su u jednostruko ulančanu listu. Zahtev ima sledeću strukturu:

```
struct IORequest {
  REG* buffer; // Data buffer (block)
  unsigned int size; // Buffer (blok) size
  int status; // Status of operation
  IORequest* next; // Next in the list
};
```

Na prvi zahtev u listi pokazuje globalni pokazivač `ioHead`. Kada u praznu listu kernel stavi prvi zahtev, pozvaće operaciju `transfer` koja treba da pokrene prenos za taj prvi zahtev i pokrene DMA kontroler upisom u bit *Start*. Kada se završi prenos zadat jednim zahtevom, potrebno je u polje `status` date strukture preneti status završene operacije (0 – ispravno završeno do kraja, -1 – greška), izbaciti obrađeni zahtev iz liste i pokrenuti prenos za sledeći zapis u listi, bez zaustavljanja DMA kontrolera. Ako zahteva u listi više nema, tek onda treba zaustaviti DMA kontroler resetovanjem bita *Start*. Kada bude stavljao novi zahtev u listu, kernel će proveriti i videti da je ona bila prazna, pa ponovo pozvati operaciju `transfer` itd.

Potrebno je napisati kod operacije `transfer()`, zajedno sa odgovarajućom prekidnom rutinom `dmaInterrupt()` za prekid od DMA kontrolera.

```
void transfer ();
interrupt void dmaInterrupt ();
```

Rešenje:

4. (10 poena)

Da bi smanjio internu fragmentaciju, a povećao efikasnost, neki fajl sistem za alokaciju sadržaja fajla koristi klastere (*cluster*) različite veličine. Klaster uvek sadrži susedne blokove na disku (blokove sa susednim brojevima). Za sadržaj fajla pre njegovog kraja koriste se klasteri veličine `CLUSTER_SIZE` blokova, gde je `CLUSTER_SIZE` predefinisana konstanta, mali prirodan broj veći od 1. Za sam kraj sadržaja fajla koristi se samo onoliko susednih blokova na disku koliko je potrebno, odnosno klaster veličine manje od ili jednake `CLUSTER_SIZE` blokova.
Sistem primenjuje indeksiranu alokaciju sadržaja fajla, pri čemu je indeks u samom FCB. Polje `index` strukture FCB predstavlja indeks kao niz, pri čemu svaki element *i* sadrži broj prvog fizičkog bloka (početak klastera) sa sadržajem fajla (ili 0, ako je neiskorišćen) za klaster sa rednim brojem *i* (počev od 0). Polje `size` u FCB sadrži veličinu sadržaja fajla u bajtovima. Veličina bloka na disku u bajtovima je `BLOCK_SIZE`. FCB je uvek učitan u memoriju za otvoren fajl.

Realizovati funkciju `getPBlock` :

```
size_t getPBlock (FCB* fcb, size_t bt);
```

Ova funkcija treba da vrati broj fizičkog bloka za bajt sa zadatim logičkim rednim brojem `bt` unutar sadržaja fajla (0 u slučaju da `bt` prekoračuje veličinu sadržaja fajla).

Rešenje: