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, jul 2020.
Datum: 13. 7. 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)

Dat je neki veliki jednodimenzioni niz `data` veličine `N`, čiji su elementi tipa `Data` (deklaracije dole). Jedan element niza obrađuje procedura `processData`. Ovaj niz potrebno je obraditi pomoću `n` uporednih niti (procedura `parallelProcessing`), gde je `n` zadati parametar, tako što svaka od tih `n` uporednih niti iterativno obrađuje približno isti broj elemenata ovog velikog niza. Drugim rečima, niz treba particionisati na `n` disjunktnih podnizova, što približnije jednakih, i te particije obrađivati uporedo, kako bi se niz obradio paralelno na višeprocesorskom sistemu.

Koristiti sistemski poziv `thread_create` koji kreira nit nad funkcijom na koju ukazuje prvi parametar, i pozivu te funkcije dostavlja drugi parametar ovog sistemskog poziva. Ova funkcija vraća identifikator kreirane niti (pozitivna vrednost), odnosno status greške (negativna vrednost). Ignorisati moguće greške.

```
const int N = ...;
class Data;
Data data[N];
void processData(Data*);
void parallelProcessing (int n);

int thread_create (void (*thread_body)(void*), void* arg);
```

Rešenje:

2. (10 poena)

Neki sistem koristi straničnu organizaciju memorije. Logičke segmente koje je proces alocirao sistem opisuje strukturama tipa `SegDesc` u dvostruko ulančanoj listi za svaki proces. Pritom se za svaki proces pri kreiranju uvek alociraju najmanje dva logička segmenta koji pokrivaju najniže i najviše adrese virtuelnog adresnog prostora procesa (preslikavaju se u prostor koji koristi kernel). Logički segment je uvek poravnat i zaokružen na stranice. U strukturi `SegDesc` polja `next` i `prev` služe za ulančavanje u listu, polje `pg` sadrži broj prve stranice logičkog segmenta, a polje `sz` sadrži veličinu segmenta izraženu u broju stranica. Tip `size_t` je neoznačen celobrojni tip dovoljno velik da predstavi veličinu virtuelnog adresnog prostora. Konstanta `PAGE_SZ` predstavlja veličinu stranice u bajtovima (adresibilnim jedinicama), a konstanta `PAGE_OFFS_SZ` veličinu polja u virtuelnoj adresi za broj bajta unutar stranice; obe ove konstante su tipa `size_t`. Data je i pomoćna funkcija `insert_seg_desc` koja alocira jednu strukturu `SegDesc` i ulančava je u listu kao što je pokazano.

```
struct SegDesc {
  SegDesc *prev, *next;
  size_t pg, sz;
  ...
};

int insert_seg_desc (SegDesc* sd, size_t pg, size_t sz) {
  SegDesc* nsd = alloc_seg_desc();
  if (!nsd) return -1;  // Error: cannot allocate SegDesc
  nsd->pg = pg; nsd->sz = sz;
  nsd->next = sd->next; nsd->prev = sd;
  sd->next->prev = nsd; sd->next = nsd;
  return 0;
}
```

Implementirati internu funkciju kernela `vmalloc` koja treba da alocira logički segment kako je proces tražio. Traženi logički segment je proizvoljne veličine `size` u bajtovima (iako ovaj parametar ne mora biti zaokružen, segment se uvek alocira kao ceo broj stranica). Ukoliko je parametar `addr` različit od *null*, pokušava se alokacija počev od stranice kojoj pripada ova adresa; ukoliko tu nije moguće alocirati segment tražene veličine, funkcija odbija alokaciju i treba da vrati grešku (vrednost *null*). Ukoliko je parametar `addr` jednak *null*, sistem treba da pokuša alokaciju segmenta bilo gde gde je to moguće; ukoliko nije moguće, treba da vrati *null*. U slučaju uspeha, u oba slučaja ova funkcija treba da vrati adresu početka segmenta (poravnatu na stranicu). Prvi parametar je glava liste struktura `SegDesc` procesa koji je tražio ovu uslugu (sigurno različito od *null*).

```
void* vmalloc (SegDesc* head, void* addr, size_t size);
```

Rešenje:

3. (10 poena)

Realizovati u potpunosti klasu `DoubleBuffer` čiji je interfejs dat. Ova klasa implementira dvostruki bafer. Proizvođač stavlja u „izlazni“ bafer blokove veličine `chunkSizeProd` znakova pozivom operacije `put()`; potrošač uzima iz „ulaznog“ bafera blokove veličine `chunkSizeCons` znakova pozivom operacije `get()`. Kada obojica završe sa svojim baferom, baferi zamenjuju uloge. Proizvođač i potrošač su uporedne niti (ne treba ih realizovati), dok je sva potrebna sinhronizacija unutar klase `DoubleBuffer`. Pretpostaviti da je zadata veličina oba bafera u znakovima (argument `size` konstruktora) celobrojan umnožak obe zadate veličine bloka (argumenti `chunkSize`). Za sinhronizaciju koristiti semafore školskog jezgra.

```
class DoubleBuffer {
public:
  DoubleBuffer (size_t size, size_t chunkSizeProd, size_t chunkSizeCons);
  void put (const char* buffer);
  void get (char* buffer);
private:
  ...
};
```

Rešenje:

4. (10 poena)

Dva procesa, pošiljalac i primalac, komuniciraju slanjem, odnosno prijemom „poruke“ preko fajla kao deljenog objekta. Kada pripremi poruku određene veličine u svom baferu, pošiljalac kreira fajl sa nazivom „*buffer.bin*“ u roditeljskom direktorijumu svog tekućeg direktorijuma, upiše poruku u taj fajl i zatvori fajl. Kada primalac „vidi“ ovaj fajl, on iz njega pročita poslatu poruku, a potom obriše taj fajl. Zbog ovakve sinhronizacije, pošiljalac treba da funkcioniše ovako: ukoliko pri pokušaju kreiranja navedenog fajla zaključi da taj fajl već postoji (jer ga primalac još uvek nije obrisao), pošiljalac se uspavljuje 5 sekundi, ostavljajući priliku primaocu da pročita i obriše fajl, a onda pokušava ponovo. Oba procesa izvršavaju se u ime istog korisnika. Napisati kod funkcije

```
int send(const void *buffer, size_t size);
```

procesa pošiljaoca koju on poziva kada želi da pošalje poruku datu u baferu zadate veličine. Ukoliko dođe do bilo koje greške koju ne može da obradi, ova funkcija treba da vrati -1, a u slučaju uspeha vraća 0. Na raspolaganju su sledeći standardni POSIX sistemski pozivi:

- `unsigned int sleep(unsigned int sleep_time_in_seconds)`: uspavljuje (suspenduje) pozivajući proces na zadato vreme (u sekundama);
- `int close (int fd)`: zatvara fajl sa zadatim deskriptorom;
- `int write (int fd, const void *buffer, size_t size)`: upisuje dati sadržaj u fajl sa zadatim deskriptorom;
- `int open(const char *pathname, int flags, mode_t mode)`: otvara fajl sa zadatom putanjom; argument *flags* mora da uključi jedno od sledećih prava pristupa: *O_RDONLY, O_WRONLY,* ili *O_RDWR.* Ako je uključen i fleg *O_CREAT*, fajl će biti kreiran ukoliko ne postoji; ako je pritom uključen i *O_EXCL*, a fajl već postoji, funkcija će vratiti grešku (-1), a kod greške postavljen u globalnoj sistemskoj promenljivoj *errno* biće jednak *EEXIST*. Argument *mode* definiše prava pristupa za fajl koji se kreira samo u slučaju da je uključen *O_CREAT*, i to na sledeći način:

S_IRWXU	*0x0700 user (file owner) has read, write and execute permission*
S_IRUSR	*0x0400 user has read permission*
S_IWUSR	*0x0200 user has write permission*
S_IXUSR	*0x0100 user has execute permission*
S_IRWXG	*0x0070 group has read, write and execute permission*
S_IRGRP	*0x0040 group has read permission*
S_IWGRP	*0x0020 group has write permission*
S_IXGRP	*0x0010 group has execute permission*
S_IRWXO	*0x0007 others have read, write and execute permission*
S_IROTH	*0x0004 others have read permission*
S_IWOTH	*0x0002 others have write permission*
S_IXOTH	*0x0001 others have execute permission*

Rešenje: