
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: Drugi, avgust 2026.
Datum: 6. 9. 2026.

Drugi kolokvijum iz Operativnih sistema 1

Kandidat: _____

Broj indeksa: _____ *E-mail:* _____

Kolokvijum traje 90 minuta. Dozvoljeno je korišćenje literature.

Zadatak 1 _____/10

Zadatak 2 _____/10

Ukupno: _____/20 = _____% = _____/10

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)

Neki operativni sistem u svom jezgri procese i niti generalizuje pojmom *zadatak* i predstavlja objektima klase `Task`. Svaki zadatak koristi svoj stek tokom izvršavanja koda jezgra; na tom steku se čuva njegov kontekst. Za red spremnih zadataka zadužena je klasa `Scheduler` koja realizuje algoritam raspoređivanja na osnovu prioriteta, tako što se uvek izvršava izvršiv zadatak sa najvišim prioritetom.

Klasa `ContextSwitch`, čija je jedina instanca dostupna pozivom njene statičke funkcije članice `instance`, čuva u sebi pokazivač `oldRunning`, koji ukazuje na zadatak koji se trenutno izvršava, tj. izvršavao se pre ulaska u jezgro zbog obrade sistemskog poziva ili prekida, kao i pokazivač `newRunning` koji je po ulasku u jezgro postavljen na istu tu vrednost, a onda, tokom obrade, menja svoju vrednost po potrebi tako da ukazuje na zadatak koji je kandidat da bude sledeći tekući po završetku svih promena i cele obrade. Da se ne bi svaki put menjao kontekst kad se promeni `newRunning`, što se može dogoditi više puta tokom obrade sistemskog poziva, funkcija `pendYield` samo postavi fleg koji ukazuje na to da promenu konteksta treba uraditi, pa kada jezgro završi celu obradu i potencijalno više puta promeni vrednost `newRunning`, na samom kraju obrade promeni kontekst sa `oldRunning` na `newRunning`.

Dole su date implementacije dve nestatičke funkcije članice klase `ContextSwitch`:

- `suspendRunning`: poziva je jezgro kada tekući zadatak treba suspendovati; tekući zadatak treba da bude stavljen u odgovarajući red čekanja pre poziva ove funkcije;
- `preemptRunning(Task*)`: poziva je jezgro kada se pojavio (aktivirao) novi izvršiv (spreman) zadatak koji je dat kao argument i zato procesor možda treba preuzeti od tekućeg zadatka (ako je ovaj novi višeg prioriteta);
- `getRunning()`: vraća `oldRunning`.

One se oslanjaju na sledeće funkcije klase `Scheduler`:

- `Scheduler::get`: iz reda spremnih bira jedan zadatak po algoritmu raspoređivanja, izbacuje ga iz tog reda i vraća ga pozivaocu; statička operacija klase `Scheduler`;
- `Scheduler::select`: od dva zadatka koji su dati kao argumenti vraća onaj sa višim prioritetom, a onaj drugi smešta u red spremnih; ukoliko su ta dva istog prioriteta, vraća ovaj prvi; statička operacija klase `Scheduler`.

Korišćenjem ovih operacija implementirati funkcije `Semaphore::wait` i `Semaphore::signal` semafora, kao i funkciju `Timing::wakeup` koja treba da probudi sve procese koji su uspavani na vreme. Za klasu `Semaphore` usvojiti pretpostavke kao na predavanjima, a za klasu `Timing` pretpostaviti sledeće:

- ima podatak član `now` koji sadrži tekuće realno apsolutno vreme u nanosekundama proteklo od nekog referentnog trenutka;
- ima podatak član `sleeping` koji sadrži red uspavanih zadataka, uređen po rastućem apsolutnom trenutku buđenja; ovaj red ima funkcije za dohvaćanje (`getFirst`) i uklanjanje (`removeFirst`) prvog zadatka iz reda;
- klasa `Task` ima funkciju `getWakeupTime` koja vraća apsolutni trenutak buđenja.

Sve pomenute funkcije su deo implementacije jezgra i izvršavaju se bez preotimanja, tako da ne treba brinuti o međusobnom isključenju (tj. pozivima *lock/unlock*).

```
inline void ContextSwitch::suspendRunning () {
    if (this->newRunning==this->oldRunning) {
        this->newRunning = Scheduler::get();
        this->pendYield();
    }
}

inline void ContextSwitch::preemptRunning (Task* resumed) {
    this->newRunning = Scheduler::select(this->newRunning,resumed);
    if (this->newRunning!=this->oldRunning) pendYield();
}
```

Rešenje:

2. (10 poena)

Korišćenjem POSIX sistemskih pozivia za semafore i *getpid*, implementirati apstrakciju *Mutex* koja predstavlja sinhronizacionu primitivu za međusobno isključenje, koja se može koristiti samo u te svrhe, tj. uz ograničenje da samo proces koji je zaključao kritičnu sekciju može da je otključa: ukoliko to nije zadovoljeno, operacije *exit* treba da vrati grešku (negativnu vrednost). Osim toga, treba sprečiti mrtvu blokadu procesa sa samim sobom: proces ne sme pozvati *entry* ako je taj isti proces već zaključao dati *mutex*.

Ova apstrakcija treba da ima sledeći objektno orijentisani interfejs:

```
class Mutex {
public:
    Mutex (const char *name);
    int entry ();
    int exit();
};
```

Rešenje: