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, maj 2026.
Datum: 31. 5. 2025.

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 jezgru 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. Na raspolaganju su sledeće operacije čiji su potpisi dati dole:

- `getRunning`: vraća tekući zadatak;
- `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`;
- `Scheduler::putGet`: poredi prioritet zadatka koji je dat kao argument sa prioritetom najprioritetnijeg zadatka u redu spremnih i vraća jedan od ta dva sa višim prioritetom, a onaj drugi smešta u red spremnih; ukoliko su ta dva zadatka istog prioriteta, zadatak koji je dat kao argument stavlja u red spremnih, a onaj iz reda spremnih izbacuje iz tog reda i vraća ga; statička operacija klase `Scheduler`;
- `yield`: obavlja promenu konteksta prelaskom sa izvršavanja prvog datog zadatka na izvršavanje drugog datog zadatka, čuvanjem konteksta zadatka na njegovom steku.

Korišćenjem ovih operacija implementirati sledeće funkcije čiji su potpisi dati dole, a koje jezgro poziva u odgovarajućim situacijama u kojima (možda) treba konačno uraditi promenu konteksta nakon što su već završene ostale potrebne radnje u jezgru; promenu konteksta raditi samo ako je to stvarno potrebno, tj. ako se tekući proces zaista menja:

- `suspendRunning`: poziva je jezgro kada tekući zadatak treba suspendovati; tekući zadatak je već stavljen u neki 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);
- `preemptRunning()`: poziva je jezgro kada je tekućem zadatku istekao dodeljeni vremenski odsečak i kada procesor treba dati drugom spremnom zadatku istog prioriteta, ako takvog ima.

```
Task* getRunning();
Task* Scheduler::get();
Task* Scheduler::select(Task* oldRunning, Task* resumed);
Task* Scheduler::putGet(Task* oldRunning);
void yield(Task* oldRunning, Task* newRunning);

void suspendRunning ();
void preemptRunning (Task* resumed);
void preemptRunning ();
```

Rešenje:

2. (10 poena)

Dole je data nepotpuna implementacija kružnog bafera kapaciteta *N* elemenata koji dozvoljava proizvođačima da u bafer smeštaju elemente neograničeno, tako što se elementi koji se stavljaju kada u baferu već ima *N* elemenata prepisuju preko elemenata koji su najdavnije u njega stavljeni (po FIFO redosledu).

Dopuniti datu implementaciju potrebnom sinhronizacijom pomoću semafora koja omogućava uporedno korišćenje ovog bafera od strane više proizvođača i potrošača. Potrošač ne može uzimati element iz praznog bafera.

```
const int N = ...;  // Capacity of the buffer
class Data;

class UnboundedBuffer {
public:
  UnboundedBuffer ();

  void  append (Data*);
  Data* take ();

private:
  Data* buffer[N];
  int head, tail, count;
};

UnboundedBuffer::UnboundedBuffer () :
   head(0), tail(0), count(0) {}

void UnboundedBuffer::append (Data* d) {
  buffer[tail] = d;
  tail = (tail+1)%N;
  if (count<N) {
    count++;
  } else {
    head = tail;
  }
}

Data* UnboundedBuffer::take () {
  Data* d = buffer[head];
  head = (head+1)%N;
  count--;
  return d;
}
```

Rešenje: