Elektrotehnički fakultet u Beogradu
Katedra za računarsku tehniku i informatiku

Predmet: Operativni sistemi 2
Nastavnik: prof. dr Dragan Milićev
Odsek: Softversko inženjerstvo, Računarska tehnika i informatika
Kolokvijum: Drugi, januar 2026.
Datum: 17. 1. 2026.

Drugi kolokvijum iz Operativnih sistema 2

Kandidat: _____

Broj indeksa: _____ *E-mail:*_____

Kolokvijum traje 90 minuta. Dozvoljeno je korišćenje literature.

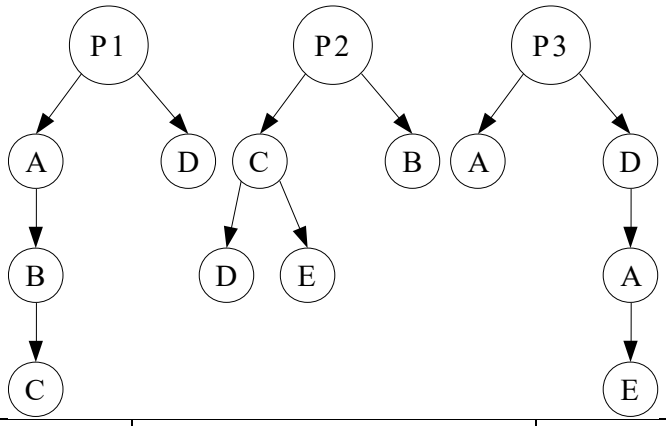
Zadatak 1 _____/10 *Zadatak 3* _____/10
Zadatak 2 _____/10

Ukupno: _____/30 = _____%

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) Mrtva blokada

Neki konkurentan program sastoji se od uporednih procesa koji poseduju ugnežđene kritične sekcije zaštićene blokirajućim međusobnim isključenjem kao što je simbolički prikazano dole. Za taj program moguće je izvršiti statičku analizu korišćenja kritičnih sekcija. Rezultat takve analize je struktura podataka oblika šume stabala, kakva je data dole za ovaj primer. Pronaći i precizno objasniti *sve* scenarije ulaska u mrtvu blokadu ovog programa i prikazati grafove zauzeća resursa za te situacije.




```
process P1 {
  mutex A {
    mutex B {
      mutex C {}
    }
  }
  mutex D {}
}
```

```
process P2 {
  mutex C {
    mutex D {}
    mutex E {}
  }
  mutex B {}
}
```

```
process P3 {
  mutex A {}
  mutex D {
    mutex A {
      mutex E {}
    }
  }
}
```

Rešenje:

2. (10 poena) Upravljanje memorijom

Dat je fajl čiji je sadržaj binaran i sastoji se od sledećeg. Na početku sadržaja fajla je zapisan jedan broj tipa `unsigned` koji predstavlja „ključ“. Odmah iza toga zapisan je niz struktura tipa `Node` u istom binarnom formatu u kom se predstavljaju i u memoriji; veličina ovog niza sigurno nije veća od `MAX_ARRAY_SIZE`. U tom nizu zapisani su čvorovi binarnog indeksnog stabla. Koren tog stabla je u indeksu 0 tog niza. Svaki čvor u polju `lchild` strukture `Node` sadrži indeks čvora levog deteta, a u polji `rchild` indeks čvora desnog deteta; u polju `val` je vrednost pridružena čvoru. U levom podstablu datog čvora čvorovi sadrže manje, a u čvorovima desnog podstablu veće vrednosti od one pridružene datom čvoru.

Korišćenjem tehnike memorijski preslikanih fajlova, napisati program koji se poziva sa jednim argumentom, nazivom fajla čiji je sadržaj u opisanom formatu, i koji treba da pronađe dati zapisani ključ u datom zapisanom indeksnom stablu i na stadardni izlaz ispiše poruku o tome da li je vrednost pronađena ili nije. Na raspolaganju su sledeći POSIX sistemski pozivi sa izvodom iz dokumentacije:

```
#include <fcntl.h>
int open (const char *pathname, int flags);
int close(int fd);
```

The open() *system call opens the file specified by* pathname*. If the specified file does not exist, it may optionally (if* O_CREAT *is specified in* flags*) be created by* open()*.*
The return value of open() *is a file descriptor, a small, nonnegative integer that is used in subsequent system calls to refer to the open file. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.*
The argument flags *must include one of the following access modes:* O_RDONLY*,* O_WRONLY*, or* O_RDWR*. These request opening the file read-only, write-only, or read/write, respectively.*
open() *returns the new file descriptor, or -1 if an error occurred.*

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t
offset);
int munmap(void *addr, size_t length);
```

mmap() *creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in* addr*. The* length *argument specifies the length of the mapping (which must be greater than 0).*
If addr *is* NULL*, then the kernel chooses the (page-aligned) address at which to create the mapping; this is the most portable method of creating a new mapping. If* addr *is not* NULL*, then the kernel takes it as a hint about where to place the mapping*[...] *The address of the new mapping is returned as the result of the call.*
The contents of a file mapping are initialized using length *bytes starting at offset* offset *in the file (or other object) referred to by the file descriptor* fd*.* offset *must be a multiple of the page size as returned by* sysconf(_SC_PAGE_SIZE)*.*
After the mmap() *call has returned, the file descriptor,* fd*, can be closed immediately without invalidating the mapping.*
The prot *argument describes the desired memory protection of the mapping (and must not conflict with the open mode of the file). It is either* PROT_NONE *or the bitwise OR of one or more of the following flags:*

PROT_EXEC *Pages may be executed.*
PROT_READ *Pages may be read.*
PROT_WRITE *Pages may be written.*
PROT_NONE *Pages may not be accessed.*

On success, mmap() *returns a pointer to the mapped area. On error, the value* MAP_FAILED *(that is,* (void *)-1*) is returned.*
The munmap() *system call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references. The region is also automatically unmapped when the process is terminated. On the other hand, closing the file descriptor does not unmap the region.*

Izostvaljajući detalje, argument *flags* u pozivu *mmap()* treba postaviti na *MAP_SHARED*.

Rešenje:

3. (10 poena) Upravljanje memorijom

Posmatra se neki alokator memorije za potrebe jezgra na principu „parnjaka“ (*buddy*). Najmanja jedinica alokacije je stranica, a ukupna raspoloživa memorija kojom može da upravlja alokator ima 2^{N-1} stranica koji su označeni brojevima $0..2^{N-1}-1$, gde je N predefinisana konstanta. Za potrebe evidencije slobodnih komada memorije, alokator vodi strukturu čija je deklaracija:

```
const int N = …;
struct buddy_t {
  uint64_t array_size;
  int64_t buddy[N];
};
```

Svaki ulaz *i* (*i*=0..*N*-1) niza `buddy` sadrži glavu liste slobodnih komada memorije veličine 2^i susednih stranica. Glava liste sadrži broj prve stranice u slobodnom komadu, a broj naredne stranice u listi je upisan na početku svake slobodne stranice u listi (-1 za kraj liste). Polje `array_size` sadrži broj ulaza u nizu `buddy` koji se koriste.

Veličine virtuelne adrese je 64 bita. Stranice kojima može da upravlja alokator su u najvišem delu adresnog prostora. Veličina stranice je 4KB. Implementirati funkcije `get_page_address` i `increase_memory`. Prva funkcija na osnovu broja stranice u alokatoru vraća pokazivač na početak te stranice, dok druga funkcija duplira veličinu prostora kojom upravlja alokator. Pre poziva funkcije `increase_memory`, nove stranice kojima će upravljati alokator su već prisutne u adresnom prostoru. Tražene funkcije treba da vrate kod greške u slučaju da tražene operacije ne mogu da se izvrše. Na počeku rada sistema alokator nema ni jednu stranicu kojom upravlja i svako proširenje prostora kojim upravlja alokator se radi pomoću tražene funkcije `increase_memory`.

Deklaracije traženih funkcija su:

```
void* get_page_address(struct buddy_t buddy, uint64_t page_num);
int increase_memory(struct buddy_t buddy);
```

Rešenje: