Sveučilište J. J. Strossmayera u Osijeku

Fakultet elektrotehnike, računarstva i informacijskih tehnologija

Kneza Trpimira 2b

HR-31000 Osijek

www.ferit.unios.hr

Laboratorijska vježba 8:

Kontrola istovremenosti kod višenitnih programa: sinkronizacijski mehanizmi i upravljanje zastojima

Sadržaj

I. UVOD

U jednoprocesorskim računalnim sustavima koji podržavaju istovremeni rad više procesa (engl. *multiprogramming systems*), procesi se izmjenjuju u registrima procesora kako bi se stekao dojam istovremenog izvođenja. Iako se na taj način ne ostvaruje stvarno paralelno izvođenje, izmjena procesa stvara dojam prividne paralelnosti i poboljšava korisničko iskustvo. S druge strane, u višeprocesorskim ili višejezgrenim računalnim sustavima moguće je postići stvarno paralelno izvođenje različitih procesa ili niti.

Na prvi pogled, očite su prednosti paralelizacije programa raspodjelom posla na nekoliko niti ili procesa. Teoretski, brzina izvođenja programa se smanjuje s faktorom koji je jednak broju niti koje izvršavaju zadani posao. U stvarnosti je to ubrzanje ipak nešto manje. Nadalje, resursi procesora su bolje iskorišteni kada postoji više manjih niti koje je moguće izmjenjivati u registrima. Dok jedna nit čeka dovršetak U/I operacije, procesor može izvoditi drugu nit. Općenito, načelo „podijeli i vladaj" smanjuje složenost programa izdvajanjem aktivnosti u zasebne niti. Međutim, različite niti upravljaju različitim elementima sustava pa im je stoga nužno dodijeliti različite prioritete i načine raspoređivanje. Iako su benefiti višenitnog programiranja neosporivi, ono također uzrokuje i određene poteškoće. Osim što je takav način programiranja složen i podložan pogreškama, potrebni su odgovarajući mehanizmi sinkronizacije i komunikacije.

U ovoj laboratorijskoj vježbi objasnit ćemo probleme kontrole istovremenosti koji se javljaju u višenitnim aplikacijama te sinkronizacijske mehanizme koji rješavaju takve probleme.

1.1. Razlika između međusobnog isključivanja i sinkronizacije

Osnovni preduvjet ostvarivanja istovremenosti je sposobnost nametanja **međusobnog isključivanja** - izuzimanje svih ostalih procesa iz tijeka akcije dok se jedan proces izvršava. Osiguravanjem međusobnog isključivanja, sprječava se nastanak uvjeta utrke. Drugim riječima, niti jedna nit izvršavanja ne može pristupiti kritičnom odsječku u vrijeme kad ga izvodi konkurentna nit. Međusobno isključivanje je moguće postići i hardverskim i softverskim putem. U prošloj laboratorijskoj vježbi analizirali smo nekoliko algoritama koji postižu međusobno isključivanje čekanjem (engl. *busy waiting*). To je tehnika u kojoj proces periodički (unutar petlje) provjerava je li uvjet zadovoljen, tj. je li slobodan kritički odsječak za izvršavanje. Postoje brojna softverska rješenja koja koriste tehniku međusobnog isključivanja čekanjem, u vidu raznih algoritama: *Dekkerov* algoritam, *Petersonov* algoritam, *Lamportov* algoritam i ostali. Međutim, ti algoritmi ne rade dobro na platformama na kojima je tijek izvršavanja dinamičan i ne može se u potpunosti kontrolirati (engl. *out-of-order execution*), jer podrazumijevaju strogo navođenje redoslijeda memorijskih operacija unutar niti.
Stoga je često poželjno koristiti sinkronizacijske mehanizme za postizanje međusobnog isključivanja, koji su inkorporirani u većinu biblioteka za upravljanje nitima. Takvi mehanizmi iskorištavaju hardverska rješenja za međusobno isključivanje ako je to moguće, a u suprotnom upotrebljavaju softverska rješenja. U literaturi se pojmovi međusobno isključivanje i **sinkronizacija** zajednički uvode i objašnjavaju, iako postoji formalna razlika među njima. Međusobno isključivanje podrazumijeva samo to da dva procesa (ili niti) ne mogu istovremeno biti u istom kritičnom odsječku. Sinkronizacija proširuje međusobno isključivanje u smislu da se njome može definirati i redoslijed izvođenja dvije ili više niti. Međutim, u literaturi se mehanizmi koji osiguravaju međusobno isključivanje te mehanizmi koji provode sinkronizaciju sveobuhvatno nazivaju **sinkronizacijskim mehanizmima**. Među

njima, oni koji isključivo služe za međusobno isključivanje se još nazivaju i **mehanizmima za zaključavanje** (engl. *locking mechanisms*).

1.2. Sinkronizacijski mehanizmi

U ovom se poglavlju prikazuju mnogi primjeri sinkronizacije niti korištenjem sinkronizacijskih mehanizama (semafor, muteks, uvjetne varijable, monitor,…). Pri osmišljavanju sinkronizacije potrebno je voditi računa o nekoliko aspekata i mogućih problema. Ispravno sinkronizirati niti ili procese znači da:

- Redoslijed izvođenja mora biti istovjetan opisu u tekstu zadatka
- Međusobno isključivanje treba koristiti sredstva za koje se to zahtijeva ili je iz zadatka očito da tako treba
- Nema mogućnosti beskonačne petlje
- Nema mogućnosti potpunog zastoja
- Nema radnog čekanja
- Početne vrijednosti svih varijabli i semafora su navedene

1.2.1. Semafor

Prvi veliki doprinos u rješavanju problema uzrokovanim istovremenim programiranjem ostvario je Edsger Dijkstra 1965. godine. Predložio je fundamentalni princip za istovremeni rad: dva ili više procesa mogu surađivati pomoću jednostavnih signala, tako da se proces može prisiliti da stane s radom na određenom mjestu sve dok ne primi specifični signal. Za signalizaciju, upotrjebljavaju se različiti sinkronizacijski mehanizmi, među kojima je prvi predložen **semafor**.

Semafor je cjelobrojna varijabla na kojoj su dozvoljene tri operacije:

1. Semafor može biti *inicijaliziran* na nenegativnu vrijednost. Uobičajeno se postavlja na broj procesa koji su u utrci.
2. Operacija *semWait* dekrementira vrijednost semafora. Ako vrijednost postaje negativna, proces koji želi izvršiti tu operaciju, biva blokiran.
3. Operacija *semSignal* operacija inkrementira vrijednost semafora. Ako je vrijednost potom manja ili jednaka 0, proces koji je blokiran *semWait* operacijom biva odblokiran.

Semafori se mogu koristiti za signalizaciju na sljedeći način: jedan proces odašilje signal preko semafora pomoću *semSignal* operacije, a drugi proces izvršava operaciju *semWait* kako bi dobio signal sa semafora.

POSIX standard omogućava korištenje semafora zajedno s Pthreads API-jem, kao odvojenu biblioteku *semaphore.h*. Također, pri kompajliranju koda koji sadrži semafore, potrebno je dodati opciju ***-lpthread -lrt.*** Semafor je reprezentiran tipom podatka *sem_t*. Funkcije za inicijalizaciju, signalizaciju, čekanje i uništavanje semafora su:

```
sem_init(sem_t *sem, int pshared, unsigned int value);

int sem_post(sem_t *sem);

int sem_wait(sem_t *sem);

sem_destroy(sem_t *sem);
```

Argumenti navedenih funkcija su:

- *sem* - semafor koji se inicijalizira

- *pshared* - argument koji specificira hoće li se novostvoreni semafor dijeliti između procesa ili niti. Nenegativna vrijednost znači da se semafor dijeli između procesa, a 0 da se dijeli između niti.
- *value* - vrijednost koja će se dodijeliti inicijaliziranom semaforu.

1.2.2. Muteks

Pojednostavljena verzija semafora je **binarni semafor**, koji može poprimiti samo dvije vrijednosti - 0 i 1. Ako je vrijednost na semaforu jednaka 0, proces se blokira. Ako je vrijednost pak jednaka 1, proces nastavlja s izvršavanjem. Sinkronizacijski mehanizam čiji je koncept vrlo sličan binarnom semaforu je **muteks** (engl. *mutex*). Ključna razlika između njih je ta što je semafor signalizacijski mehanizam, a muteks mehanizam za zaključavanje (engl. *locking mechanism*). Muteks omogućuje međusobno isključivanje, tako što zahtijeva da proces u utrci koji posjeduje ključ (muteks) može nastaviti s izvršavanjem. Svi ostali koji ne posjeduju ključ, bivaju blokirani. Kod binarnog semafora, jedan proces može "zaključati" semafor (tj. postaviti vrijednost na 0), a drugi proces ga može "otključati" (tj. postaviti vrijednost na 1). Kod muteksa, isti proces koji je zaključao muteks, mora ga i otključati.

Pthreads API ima ugrađene funkcije koje omogućavaju rad s muteksima. Muteksi se mogu primijeniti samo na niti unutar istog procesa, a ne rade između procesa kao semafori. Koriste se za sinkronizaciju takvih uvjeta utrke u kojima dvije ili više niti izvode operacije na istom memorijskom području, ali rezultati izvršavanja ovise o redoslijedu izvršavanja. Muteks je reprezentiran tipom podatka *pthread_mutex_t*. Funkcije za rad s muteksima u Pthreads API-ju su:

```
int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutex_mutexattr_t *attr);

int pthread_mutex_lock (pthread_mutex_t *mutex)

int pthread_mutex_unlock (pthread_mutex_t *mutex)

int pthread_mutex_destroy (pthread_mutex_t *mutex)
```

Argumenti navedenih funkcija su:
- *mutex* - muteks koji se inicijalizira
- *attr* - NULL ako nema atributa; inače zastavice iz *mutexattr_t* za inicijalizaciju

Tipični slijed korištenja muteksa je :
1) Kreiranje i inicijalizacija muteks varijable
2) Nekoliko niti pokušava zaključati muteks
3) Samo jedna nit uspijeva te postaje vlasnik muteksa
4) Nit koja je vlasnik se izvršava
5) Nit koja je vlasnik otključava muteks
6) Druga nit zadobiva vlasništvo nad muteksom i počinje se izvršavati
7) Nakon izvršavanja svih niti, muteks se uništava

1.2.3. Uvjetne varijable

Uz mutekse, još jedan sinkronizacijski mehanizam uključen u Pthreads API jesu uvjetne varijable (engl. *condition variables*). Muteksi se koriste za dozvolu ili zabranu pristupa kritičnom odsječku. Uvjetne varijable omogućuju blokiranje niti ukoliko određeni uvjeti nisu zadovoljeni. Bez uvjetnih varijabli, program bi se sastojao od niti koje neprekidno provjeravaju stanje uvjeta za pristup kritičnom odsječku. Takvo ponašanje vrlo opterećuje resurse, jer su sve niti u izvršavanju. Uvjetne varijable omogućuju provjeru uvjeta bez potrebe za kontinuiranim praćenjem. Primarne operacije nad uvjetnim varijablama su *pthread_cond_wait* i *pthread_cond_signal*. Prva operacija blokira nit koja se izvodi, a druga

operacija signalizira prestanak blokiranja. Uobičajena situacija korištenja uvjetnih varijabli je kad nit koja se izvodi mora pričekati da neka druga nit završi sa svojim poslom. Pri završetku posla, druga nit signalizira da se prva nit može odblokirati. Uvjetne varijable i muteksi se uvijek koriste zajedno. Uobičajeni uzorak korištenja je da jedna nit zaključava muteks, pa zatim čeka na uvjetnu varijablu kada ne može nastaviti s izvršavanjem (mora čekati na određeni događaj). S vremenom neka druga nit izvrši očekivani događaj, signalizira putem uvjetne varijable da je moguć nastavak izvršavanja prve niti. Sustavski poziv *pthread_cond_wait* automatski otključava muteks kojeg drži prva nit. Funkcije za rad s uvjetnim varijablama u Pthreads API-ju su:

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);

int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

int pthread_cond_signal(pthread_cond_t *cond);
```

Argumenti navedenih funkcija su:

- *cond* - uvjetna varijabla koja se inicijalizira
- *attr* - NULL ako nema atributa; inače zastavice iz *condattr_t* za inicijalizaciju
- *mutex* - muteks kojeg je zaključala nit koja poziva *wait* operaciju

Tipični slijed upotrebe uvjetnih varijabli je:

1) Definiranje globalnih varijabli za čiji pristup je potrebna sinkronizacija
2) Definiranje uvjetne varijable i pripadnog muteksa
3) Jedna nit zaključava muteks i radi posao sve dok ne mora čekati na određeni događaj
4) Nit koja je zaključala muteks poziva *cond_wait()* i čeka na izvršavanje događaja
5) Poziv *cond_wait*() automatski otključava pripadni muteks, a zaključava ga nit koja izvršava traženi događaj
6) Druga nit izvršava događaj i poziva *cond_signal()*
7) Druga nit otključava muteks
8) Prva nit prima signal druge niti, koji automatski zaključava muteks i nastavlja s izvođenjem

2. POZNATI SINKRONIZACIJSKI PROBLEMI

Kod sustava koji dozvoljavaju istovremen rad više procesa, postoji niz problema koji predstavljaju klasične primjere problema višestruke sinkronizacije. Prilikom dizajna novih sinkronizacijskih mehanizama, znanstvenici uobičajeno prikazuju njihovu upotrebu pri rješavanju tih problema.

2.1. Problem proizvođač-potrošač

Jedan od najpoznatijih sinkronizacijskih problema je **problem proizvođač-potrošač** (engl. *producer-consumer*). Problem opisuje dva procesa, proizvođača i potrošača, koji imaju zajednički međuspremnik (engl. *buffer*) fiksne veličine, koji se koristi kao red. Posao proizvođača je generiranje podataka, njihovo smještanje u međuspremnik i pokretanje iznova. Istodobno, potrošač "konzumira" podatke iz međuspremnika, uklanjajući ih jednog po jednog. Problem je osigurati da proizvođač neće pokušati dodati podatke u međuspremnik ako je pun te da potrošač neće ukloniti podatke iz praznog međuspremnika. Rješenje problema je da se proizvođač stavi u stanje spavanja kad je spremnik pun, a budi se kad potrošač ukloni barem jednu stvar iz spremnika. Na sličan način, potrošač spava dok je spremnik prazan, a budi se tek kad proizvođač postavi novu stvar u spremnik. U slučaju više takvih procesa, treba osigurati spremanje signala za buđenje i spavanje za odgovarajuće procese.

Problem proizvođač-potrošač može se riješiti koristeći razne sinkronizacijske mehanizme. Primjer 2 prikazuje problem implementiran pomoću Pthreads API-ja za programski jezik C, u kojemu su implementirane dvije niti koje predstavljaju proizvođača i potrošača. Obje niti izvršavaju beskonačnu petlju, u kojoj neprestano čekaju na odgovarajuće događaje, tj. pražnjenje ili punjenje međuspremnika. Nit proizvođač puni međuspremnik sve do njegovog maksimalnog kapaciteta, a nit potrošač ga prazni sve dok ga u potpunosti ne isprazni. Glavni program sadrži funkcije za kreiranje i izvođenje niti.

Primjer 1: Problem proizvođač-potrošač implementiran pomoću Pthreads API-ja

```
#include <stdio.h>
#include <pthread.h>

#define MAX_BUFFER_SIZE 50  //Maksimalni kapacitet buffera
int buffer = 0;         // Veličina buffera na početku

void *producer()
{
  while(1) {
    if(buffer==MAX_BUFFER_SIZE){
      sleep(1);//Ako je buffer popunjen, producer spava 1 sekundu
    } else {
      buffer++; //Ako u bufferu ima praznih mjesta, počinju se popunjavati
      printf("Proizvođač je u međuspremnik postavio %d.vrijednost\n", buffer);
    }
  }
  pthread_exit(0);
}

void *consumer()
{
  while(1) {
    if (buffer==0){
      sleep(1);//Ako je buffer prazan, consumer spava 1 sekundu
    } else {
      buffer--;//Ako buffer ima podataka, počinju se vaditi
      printf("Potrošač je iz međuspremnika izvadio %d.vrijednost\n", buffer+1);
    }
  }
  pthread_exit(0);
}

int main(void) {
  pthread_t pro, con;
  pthread_create(&pro, NULL, producer, NULL);
  pthread_create(&con, NULL, consumer, NULL);
  pthread_join(pro, NULL);
  pthread_join(con, NULL);
  return 0;
}
```

Nedostatak načina rješavanja problema, prikazanog u Primjeru 1, je što niti moraju neprestano provjeravati dostupnost međuspremnika. U ovom primjeru, proizvoljno je odabrana duljina spavanja od 1 sekunde, no moguće je da niti moraju čekati duže (ili kraće). Na taj način one nepotrebno troše računalne resurse, "budeći" se svake sekunde. Poželjno je implementirani signalizacijski mehanizam, pomoću kojeg jedna nit može otići u stanje spavanja u kojem ne troši računalne resurse, te biva probuđena kad se dogodi odgovarajući događaj.

2.1.1. Rješenje problema pomoću semafora

Semafori rješavaju problem proizvođač-potrošač. Rješenje prikazano u Primjeru 2, implementira tri semafora: semafor *full* za brojanje popunjenih mjesta u međuspremniku, semafor *empty* za brojanje praznih mjesta u spremniku i semafor *mutex* za osiguravanje da proizvođač i potrošač ne mogu istovremeno pristupiti međuspremniku. Inicijalno, *full* se postavlja na 0, *empty* je jednak maksimalnom kapacitetu međuspremnika, a *mutex* je jednak 1. Posljednji semafor *mutex* je binarni semafor, jer poprima vrijednost 0 kad nit ulazi u kritični odsječak, a 1 kad izlazi.

Na ovaj je način prikazana dvojaka uloga semafora: rješavanje problema međusobnog isključivanja i sinkronizacije. Dok se semafor *mutex* koristi za postizanje međusobnog isključivanja, semafori *full* i *empty* su potrebni za sinkronizaciju. Oni osiguravaju da proizvođač i potrošač čekaju s izvršavanjem kad je međuspremnik pun, odnosno dok je prazan. U odnosu na primjer 1, gdje je duljina čekanja fiksno postavljena na 1 sekundu, u primjeru 2 semafori dinamički određuju duljinu čekanja funkcijom *sem_wait()*.

Rješenje u primjeru 2 naizmjenično prebacuje izvršavanje između niti proizvođača i potrošača. U primjeru 1, proizvođač je prvo napunio međuspremnik do kraja, a zatim ga je potrošač počeo prazniti. U rješenju sa semaforima, čim proizvođač postavi novi podatak u međuspremnik, signalizira tu aktivnost potrošaču, koji zatim počinje s pražnjenjem međuspremnika. Primjer 2 prikazuje pojednostavljeni međuspremnik, tj. cjelobrojnu varijablu *n*. Za prikladniju implementaciju međuspremnika, umjesto cjelobrojne varijable moguće je koristiti polje podataka ili sličnu kolekciju.

Primjer 2: Sinkronizacija problema proizvođač-potrošač pomoću semafora

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#define MAX_BUFFER_SIZE 50//Maksimalna veličina buffera

sem_t mutex;//Semafor za kontrolu pristupa kritičnom odsječku
sem_t empty; //Semafor za brojanje praznih mjesta u bufferu
sem_t full;//Semafor za brojanje popunjenih mjesta u bufferu

void *producer()
{
 int n;
 while(1) {
   sem_wait(&empty);//Semafor down operacija
   sem_wait(&mutex);  //Operacija za ulazak u kritični odsječak
   sem_getvalue(&full, &n); // Dohvaćanje zadnje vrijednosti na semaforu i postavljanje u
varijablu n
   printf("Proizvođač je u međuspremnik postavio %d.vrijednost\n", n+1);
   sem_post(&mutex);  //Operacija za izlazak iz kritičnog odsječka
   sem_post(&full); //Semafor up operacija
 }
 pthread_exit(0);
}

void *consumer()
{
 int n;
 while(1) {
   sem_wait(&full);  //Semafor down operacija
   sem_wait(&mutex);  //Operacija za ulazak u kritični odsječak
   sem_getvalue(&full, &n); // Dohvaćanje zadnje vrijednosti na semaforu i postavljanje u
varijablu n
   printf("Potrošač je iz međuspremnika izvadio %d.vrijednost\n", n+1);
   sem_post(&mutex);  //Operacija za izlazak iz kritičnog odsječka
   sem_post(&empty);  //Semafor up operacija
 }
 pthread_exit(0);
}

int main(void) {
 sem_init(&mutex, 1, 1);
 sem_init(&empty, 1, MAX_BUFFER_SIZE);
 sem_init(&full, 1, 0);

 pthread_t pro, con;
 pthread_create(&pro, NULL, producer, NULL);
 pthread_create(&con, NULL, consumer, NULL);
 pthread_join(pro, NULL);
 pthread_join(con, NULL);
 return 0;
}
```

2.1.2. Rješenje problema pomoću muteksa i uvjetnih varijabli

Muteksi na jednostavan način omogućuju automatsku provjeru stanja međuspremnika, bez potrebe za beskonačnim čekanjem. Međutim, ukoliko se proizvođač ili potrošač blokiraju zbog nemogućnosti pisanja ili čitanja, potreban je mehanizam koji će signalizirati njihovo buđenje. U tu svrhu se koriste uvjetne varijable. Primjer 3 prikazuje rješenje problema proizvođač-potrošač pomoću muteksa i uvjetnih varijabli. Kad proizvođač postavi podatak u međuspremnik, mora čekati da potrošač izvadi postavljeni podatak, prije nego što opet počne puniti. Također, nakon što potrošač isprazni međuspremnik, mora čekati da proizvođač generira novi podatak. Korištenjem muteksa u primjeru, osigurava se da istovremeno samo jedna nit ima pristup međuspremniku. S druge strane, uvjetne varijable služe za slanje signala drugoj niti da je moguće pisanje ili čitanje.

Primjer 3: Sinkronizacija problema proizvođač-potrošač pomoću muteksa i uvjetnih varijabli

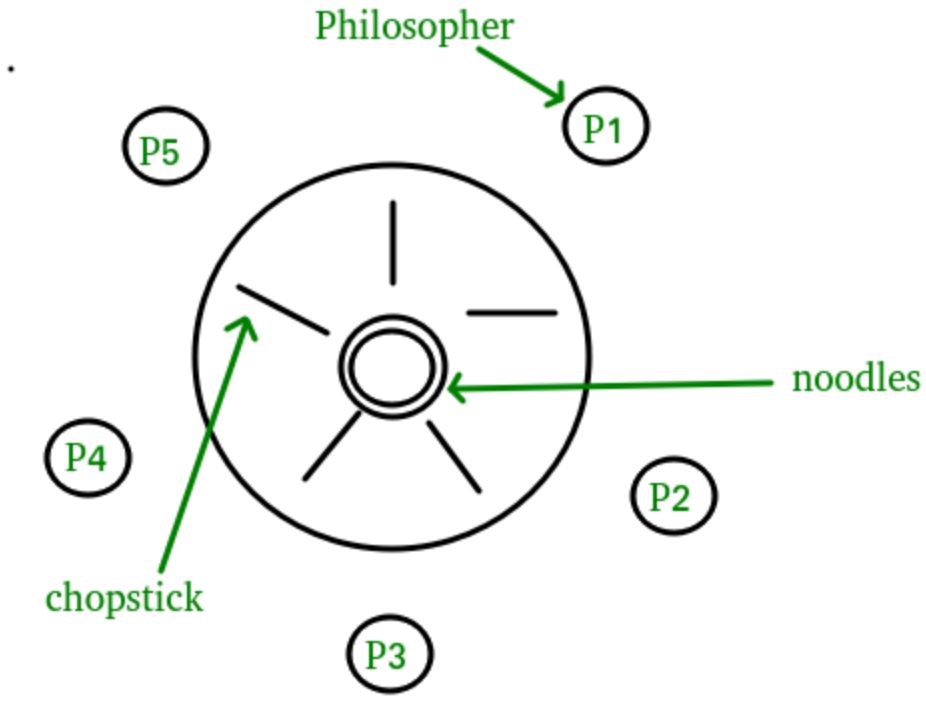
```
#include <stdio.h>
#include <pthread.h>
#define MAX_BUFFER_SIZE 50//Maksimalna veličina buffera

pthread_mutex_t mutex;
pthread_cond_t condConsumer, condProducer;
int buffer = 0;

void *producer()
{
  while(1){
    pthread_mutex_lock(&mutex);   //Ulazak u međuspremnik
    while (buffer!=0){
      pthread_cond_wait(&condProducer, &mutex); //Ako spremnik sadrži podatke, proizvođač se blokira čekajući na signal od potrošača
    }
    buffer++;
    printf("Proizvođač je u međuspremnik postavio %d.vrijednost\n", buffer);
    pthread_cond_signal(&condConsumer);
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit(0);
}

void *consumer()
{
  while(1){
    pthread_mutex_lock(&mutex);   //Ulazak u međuspremnik
    while (buffer==0){
      pthread_cond_wait(&condConsumer, &mutex); //Ako spremnik ne sadrži podatke, potrošač se blokira čekajući na signal od proizvođača
    }
    buffer--;
    printf("Potrošač je iz međuspremnika izvadio %d.vrijednost\n", buffer+1);
    pthread_cond_signal(&condProducer);
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit(0);
}

int main(void) {
  pthread_mutex_init(&mutex, 0);
  pthread_cond_init(&condConsumer, 0);
  pthread_cond_init(&condProducer, 0);
  pthread_t pro, con;
  pthread_create(&pro, NULL, producer, NULL);
  pthread_create(&con, NULL, consumer, NULL);
  pthread_join(pro, NULL);
  pthread_join(con, NULL);

  pthread_cond_destroy(&condConsumer);
  pthread_cond_destroy(&condProducer);
  pthread_mutex_destroy(&mutex);
  return 0;
}
```

2.2. Problem pet filozofa

Dobro poznati sinkronizacijski problem je **problem pet filozofa** (engl. *dining philosophers problem*). U tom problemu pet filozofa sjedi za jednim okruglim stolom kao što je prikazano na Slici 1. Ispred svakog filozofa je zdjela s hranom (koju konobari nadopunjavaju čim se isprazni). Na stolu se nalazi pet štapića, po jedan između svake zdjele. Svaki filozof cijelo vrijeme ponavlja sljedeće korake:

1. Razmišlja
2. Uzima lijevi i desni štapić ako su slobodni, inače čeka da se oslobode
3. Jede
4. Opere štapiće i vraća ih na stol, lijevo i desno od svoje zdjele

S obzirom da nema dovoljno štapića, neće svi filozofi moći istovremeno jesti. Cilj pri rješavanju ovog problema je osmisliti način sinkronizacije hranjenja među filozofima tako da svi filozofi pojedu svoj dio u razumnom vremenu. Neispravan sinkronizacijski mehanizam može rezultirati problemima kontrole istovremenosti: **zastojem** (engl. *deadlock*) ili **izgladnjivanjem** (engl. *starvation*). Prvo bi podrazumijevalo da je cjelokupni proces hranjenja blokiran jer svaki filozof čeka na akciju drugoga i tako ostaju blokirani u nedogled. Drugi problem znači da jedan od filozofa ostaje neprekidno zapostavljen i nikada ne dolazi njegov red za jelo.



Slika 1. Problem 5 filozofa

2.2.1. Implementacija problema u programskom jeziku C

Problem pet filozofa može se simulirati kao računalni sustav u kojemu su filozofi niti, a štapići kritični resursi kojima niti žele pristupiti. Proces hranjenja je kritični odsječak, tj. programski kod koji pristupa štapićima.

Primjer 4: Problem pet filozofa u programskom jeziku C

```
#include <stdio.h>
#include <pthread.h>

#define NO_OF_PHILOSOPHERS 5

pthread_t philosopher[NO_OF_PHILOSOPHERS];
int status[NO_OF_PHILOSOPHERS];

void printStatus() {
  for (int i=0; i<NO_OF_PHILOSOPHERS; i++){
    printf("Philosopher %d is", i);
    if (status[i])
      printf("eating\n");
    else
      printf("thinking\n");
  }
  printf("\n\n");
}

void *diningPhilosophers(void *id) {
  int *pid = (int *) id;
  while(1) {
    int *left_chopstick = *pid;
    int *right_chopstick = (*pid + 1) % NO_OF_PHILOSOPHERS;
    // čekanje na lijevi i desni štapić pid i (pid + 1) % NO_OF_PHILOSOPHERS
    status[*pid] = 1;
    // jednom kada je dobio lijevi i desni štapić, filozof pid je spreman za jelo i čeka da se
oslobodi zdjela
    printStatus();
    sleep(3000); // busy waiting – filozof spava XY sekundi nakon čega ponovno provjerava
je li jelo dostupno
    status[*pid] = 0;
    // nakon jela, filozof mora oprati i otpustiti štapiće
  }
}

int main(void) {
  int pid[NO_OF_PHILOSOPHERS];
  for (int i=0; i<NO_OF_PHILOSOPHERS; i++){
    pid[i] = i;   // polje od 5 filozofa, pri čemu je svaki identificiran svojim indeksom
    status[i] = 0;// na početku niti jedan filozof ne jede
  }

  for (int i=0; i<NO_OF_PHILOSOPHERS; i++){
    pthread_create(&philosopher[i], NULL, &diningPhilosophers, &pid[i]);
  }

  for (int i=0; i<NO_OF_PHILOSOPHERS; i++){
    pthread_join(philosopher[i], NULL);
  }
  return 0;
```

2.2.2. Sinkronizacija problema pet filozofa pomoću semafora

Ovaj problem je potencijalno moguće riješiti koristeći semafore kao sinkronizacijske mehanizme. Svaki štapić se može predstaviti strukturom semafora nad kojom je moguće obavljati *wait* i *post* operacije. Zdjela s hranom predstavlja kritični odsječak i može se zaštiti muteksom. Tako svaka nit (filozof) pri ulasku u beskonačnu petlju čeka na semafor (štapić) te kada se oslobodi ulazi u kritični odsječak (jede iz zdjele). Nakon završetka s jelom, nit izlazi iz kritičnog odsječka te šalje signal preko semafora.

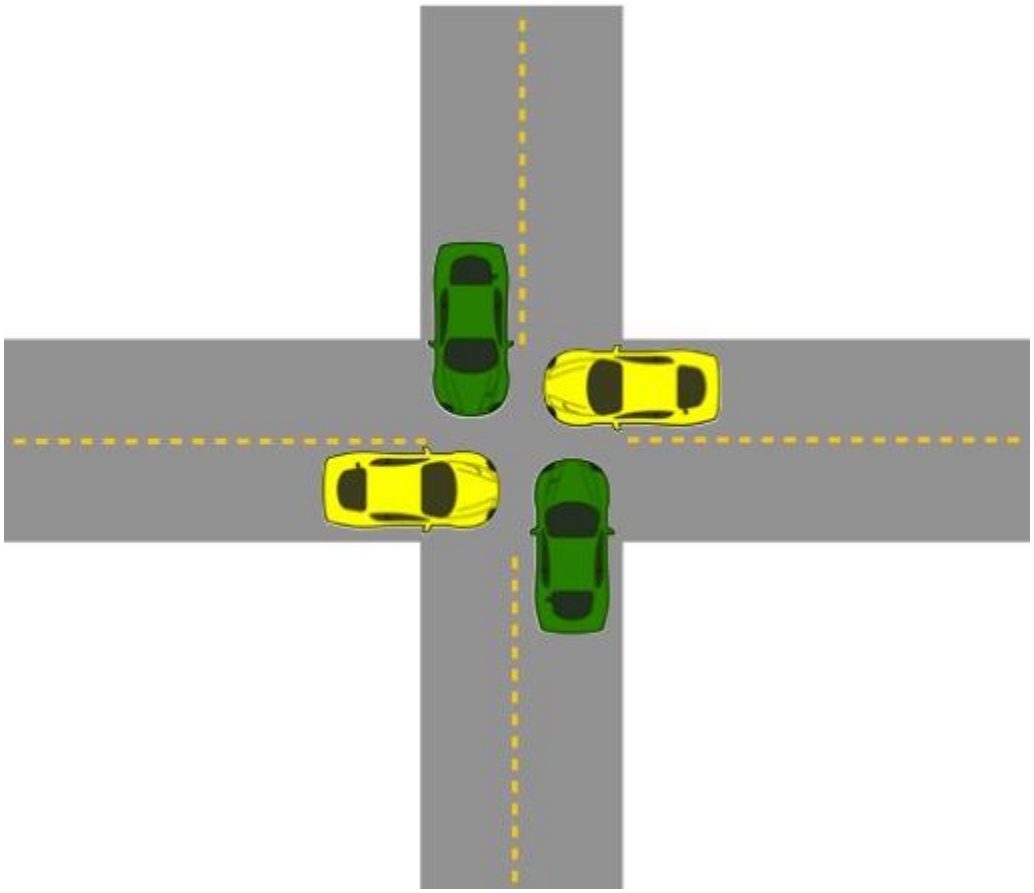
Primjer 5 prikazuje programski kod u kojem je provedena sinkronizacija problema pet filozofa upotrebom semafora. Problem s navedenim rješenjem jest u slučaju da sve niti rade paralelno te da istovremeno pozivaju operaciju *sem_wait*. S obzirom da su inicijalno svi štapići na stolu, sve niti bi prošle kroz prvi poziv operacije *sem_wait* čekajući na drugu nit. Međutim, tada bi svi semafori bili blokirani, tj. sve niti bi zapele na operaciji čekanja te bi nastao **potpuni zastoj.**

Primjer 5: Sinkronizacija problema pet filozofa upotrebom semafora

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

#define NO_OF_PHILOSOPHERS 5

sem_t chopstick[NO_OF_PHILOSOPHERS];
sem_t mutex;
pthread_t philosopher[NO_OF_PHILOSOPHERS];
int status[NO_OF_PHILOSOPHERS];

void printStatus() {
  for (int i=0; i<NO_OF_PHILOSOPHERS; i++){
    printf("Philosopher %d is ", i+1);
    if (status[i])
      printf("eating\n");
    else
      printf("thinking\n");
  }
  printf("\n\n");
}

void *diningPhilosophers(void *id) {
  int *pid = (int *) id;
  while(1) {
    sem_wait(&chopstick[*pid]); // lijevi štapić
    sem_wait(&chopstick[(*pid + 1) % NO_OF_PHILOSOPHERS]); // desni štapić
    status[*pid] = 1;
    sem_wait(&mutex);
    printStatus();
    sem_post(&mutex);
    status[*pid] = 0;
    sem_post(&chopstick[*pid]);
    sem_post(&chopstick[(*pid + 1) % NO_OF_PHILOSOPHERS]);
  }
}

int main(void) {
  int pid[NO_OF_PHILOSOPHERS];
  sem_init(&mutex, 0, 1);

  for (int i=0; i<NO_OF_PHILOSOPHERS; i++){
    pid[i] = i;   // polje od 5 filozofa, pri čemu je svaki identificiran svojim indeksom
    status[i] = 0;// na početku niti jedan filozof ne jede
  }

  for (int i=0; i<NO_OF_PHILOSOPHERS; i++){
    sem_init(&chopstick[i], 0, 1);
  }

  for (int i=0; i<NO_OF_PHILOSOPHERS; i++){
    pthread_create(&philosopher[i], NULL, &diningPhilosophers, &pid[i]);
  }

  for (int i=0; i<NO_OF_PHILOSOPHERS; i++){
    pthread_join(philosopher[i], NULL);
  }
  return 0; }
```

3. UPRAVLJANJE ZASTOJIMA

Konkurentni procesi međusobno komuniciraju na razne načine. Uslijed tih interakcija, javljaju se problemi kontrole istovremenosti: međusobno isključivanje i zastoji. **Zastoj** se može definirati kao trajno blokiranje skupa procesa koji se natječu za resurse ili međusobno komuniciraju. Skup procesa je u zastoju ako je svaki proces blokiran i čeka na događaj kojeg može pokrenuti samo drugi blokirani proces u skupu. Prikladna ilustracija zastoja u svakodnevnom životu prikazana je na Slici 2.



Slika 2. Ilustracija zastoja

3.1. Uvjeti zastoja

Četiri uvjeta moraju biti istovremeno ispunjeni, kako bi mogli utvrditi da je došlo do zastoja:

- Međusobno isključivanje - samo jedan proces smije koristiti resurs u nekom trenutku.
- Zadržavanje i čekanje (engl. *hold and wait*) - proces drži svoje alocirane resurse dok čeka na dostupnost novih resursa.
- Neprekidnost (engl. *no-preemption*) - nijedan resurs se ne može "nasilno" oduzeti procesu koji ga je alocirao.
- Kružno čekanje (engl. *circular wait*) - postoji zatvoreni lanac procesa, u kojem svaki proces drži resurse potrebne idućem procesu u lancu.

Za razliku od ostalih problema konkurentnosti, ne postoji efikasna tehnika za rješavanje zastoja. Međutim, zato se predlažu pristupi kojima se nastoji spriječiti (engl. *prevent*), detektirati (engl. *detect*) ili izbjeći (engl. *avoid*) zastoje. Strategija sprječavanja zastoja

podrazumijeva takav dizajn sustava u kojem je mogućnost zastoja isključena. S druge strane, operacijski sustav provodi detekciju zastoja tako da izvršava algoritam koji detektira uvjet kružnog čekanja. Konačno, izbjegavanje zastoja dozvoljava najveći stupanj konkurentnosti, a izvodi se dinamičkom provjerom svakog zahtjeva za resursom u smislu mogućnosti pojave zastoja. Jedna od najpoznatijih strategija za provođenje izbjegavanja zastoja je **Bankarev algoritam** (engl. *Banker's algorithm*).

3.2. Bankarev algoritam

Bankarev algoritam je strategija sprječavanja zastoja, a proizlazi iz analogije mjesnog bankara koji dodjeljuje kredit klijentima. Ideja je zasnovana na principu da bankar ne daje maksimalni iznos kredita odjednom svim klijentima, nego daje tako da može zadovoljiti isplate drugim klijentima – ne dovodi se u nesigurno stanje. U slučaju operacijskih sustava, bankarev algoritam odlučuje o dodjeljivanju resursa procesima.

Tablica 1 definira vektore i matrice potrebne za provođenje Bankarevog algoritma u slučaju n procesa i m resursa.

Tablica 1. Strukture Bankarevog algoritma

Vektor	Značenje
$\boldsymbol{R} = \begin{pmatrix} \boldsymbol{R_{11}} & \cdots & \boldsymbol{R_{1m}} \\ \vdots & \ddots & \vdots \\ \boldsymbol{R_{n1}} & \cdots & \boldsymbol{R_{nm}} \end{pmatrix}$	Tablica potrebnih resursa: R_{ij} predstavlja zahtjev procesa i za resurs j
$\boldsymbol{E} = (\boldsymbol{E_1}, \boldsymbol{E_2}, \boldsymbol{E_3}, \ldots, \boldsymbol{E_n})$	Broj instanci resursa: E_i predstavlja količinu i-tog resursa u sustavu
$\boldsymbol{C} = \begin{pmatrix} \boldsymbol{C_{11}} & \cdots & \boldsymbol{C_{1m}} \\ \vdots & \ddots & \vdots \\ \boldsymbol{C_{n1}} & \cdots & \boldsymbol{C_{nm}} \end{pmatrix}$	Tablica alociranih resursa: C_{ij} predstavlja količinu alociranih resursa j za proces i

Primjer 6: Primjer Bankarevog algoritma

Zadan je sustav sa 4 resursa čiji je broj instanci definiran vektorom **E** = [4 2 4 6]. Pretpostavimo da je u memoriji u nekom trenutku 4 procesa (A, B, C, D) sa sljedećom tablicom alociranih resursa **C** te tablicom potrebnih resursa **R**. Potrebno je provjeriti mogu li se procesi izvršiti (je li sustav rješiv) ili će doći do zastoja.

$$\mathbf{C} = \begin{pmatrix} A & 1 & 0 & 2 & 1 \\ B & 0 & 1 & 2 & 0 \\ C & 1 & 0 & 0 & 1 \\ D & 1 & 0 & 0 & 0 \end{pmatrix} \qquad \mathbf{R} = \begin{pmatrix} A & 0 & 1 & 2 & 0 \\ B & 1 & 0 & 0 & 3 \\ C & 3 & 2 & 4 & 4 \\ D & 0 & 2 & 3 & 1 \end{pmatrix}$$

Prvo računamo ukupnu zauzetost svakog resursa po svim procesima, zbrajajući retke matrice C:

$$\begin{array}{cccc} 1 & 0 & 2 & 1 \\ 0 & 1 & 2 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \quad + \\ - & - & - & - \quad - \\ 3 & 1 & 4 & 2 \end{array}$$

Zatim oduzimamo od broj instanci resursa ukupnu zauzetost svakog resursa:

$$\begin{array}{lcccc} \mathbf{E} = & (4 & 2 & 4 & 6) \\ - & (3 & 1 & 4 & 2) \\ \hline \mathbf{A} = & (1 & 1 & 0 & 4) \end{array}$$

Nadalje, redom se dodjeljuju resursi onim procesima čiji je zahtjeve moguće ispuniti. Promatramo redak u matrici **R** koji ima sve elemente manje ili jednake vektoru **A**. Po pronalasku takvog reda, uzimamo ekvivalentni red iz matrice **C** i nadodajemo vektoru **A**. Ukoliko nije moguće naći odgovarajući red u matrici **R** kojemu su elementi manji ili jednaki od elemenata vektora **A**, zaustavljamo algoritam i pišemo da je došlo do zastoja!

$$\begin{array}{lcccc} \mathbf{A} = & (1 & 1 & 0 & 4) \\ + & (0 & 1 & 2 & 0) \end{array}$$ Gledamo 2. redak matrice R, a dodajemo 2. redak matrice C

$$\begin{array}{lcccc} & (1 & 2 & 2 & 4) \\ + & (1 & 0 & 2 & 1) \end{array}$$ Gledamo 1. redak matrice R, a dodajemo 1. redak matrice C

$$\begin{array}{lcccc} & (2 & 2 & 4 & 5) \\ + & (1 & 0 & 0 & 0) \end{array}$$ Gledamo 4. redak matrice R, a dodajemo 4. redak matrice C

$$\begin{array}{lcccc} & (3 & 2 & 4 & 5) \\ + & (1 & 0 & 0 & 1) \end{array}$$ Gledamo 3. redak matrice R, a dodajemo 3. redak matrice C

$$\begin{array}{lcccc} & (4 & 2 & 4 & 6) \end{array}$$

Dobiveni rezultat je jednak početnom vektoru **E**, što znači da je sustav rješiv i neće doći do zastoja.

3.3. Monitori

Problemi sa semaforima (npr. potpuni zastoj) nastaju u sustavima s više dretvi i više sredstava ili složenijim problemima sinkronizacije gdje semafori nisu dovoljni. Tony Hoare je 1970-tih predložio drugačiji mehanizam sinkronizacije kojeg je nazvao **monitor**. Svojstva ovog sinkronizacijskog mehanizma su:

- Sve kritične radnje koje uključuju zajednička sredstva se moraju obavljati u kontroliranom okruženju – monitoru
- Monitori su vrlo slični kritičnim odsječcima, ali nisu sinkronizacijski mehanizmi na razini jezgre već na razini korisnika
- Monitor ima proširenu funkcionalnosti (dodatne operacije osim uđi i izađi)
- U monitoru se mijenjaju i ispituju varijable koje opisuju stanje sustava
- Ako stanje sustava nije povoljan za nastavak rada niti, ona se blokira i privremeno napušta monitor
- Blokiranu nit može odblokirati druga nit
- U monitoru je uvijek najviše aktivna jedna nit
- Moglo bi se reći da je monitor proširenje binarnog semafora

Za razliku od prijašnjih sinkronizacijskih mehanizama (muteks, semafor, uvjetne varijable), monitor nije unaprijed definirana struktura podatka koju je moguće naći kao dio sinkronizacijskih biblioteka. Prvo navedeno svojstvo monitora upravo navodi da se sve kritične radnje obavljaju unutar monitora, što znači da je to korisnički definirana struktura podataka koja obuhvaća procedure koje više niti izvode istovremeno (npr. kod problema pet filozofa to su procedure *eat*, *think*, *take_chopstick*, *put_chopstick*...).

Problem pet filozofa se može riješiti pomoću monitora tako da se izbjegne totalni zastoj. Primjer 7 prikazuje idejno rješenje sinkronizacije problema pet filozofa pomoću proizvoljne strukture monitor.

Primjer 7: Idejno rješenje sinkronizacije problema pet filozofa upotrebom monitora

```
nit Filozof (i)
{
   ponavljaj {
      misli
      uzmi_štapiće (i)  // monitorska funkcija
      jedi
      vrati_štapiće (i)  // monitorska funkcija
  }
}
m-funkcija uzmi_štapiće (i)
{
   dok (oba štapića oko filozofa i nisu slobodna)
      blokiraj_nit

   uzmi oba štapića oko filozofa i
}

m-funkcija vrati_štapiće (i)
{
   vrati oba štapića oko filozofa i
   propusti desnog filozofa ako čeka
   propusti lijevog filozofa ako čeka
}
```

4. ZADACI ZA SAMOSTALNI RAD

Pristupiti poslužitelju linux.etfos.hr pomoću Putty SSH klijenta. Sljedeće zadatke riješite koristeći uređivač teksta nano, sintaksu programskog jezika C i Bash ljusku operacijskog sustava Linux. Za kompajliranje programa napisanih u C jeziku koristite GCC kompajler koji je već instaliran na poslužitelju. **Kod kompajliranja ne zaboravite dodati opciju -lpthread.** Za svaki zadatak kopirajte programski kod, naredbe ljuske i dobiveni izlaz u Zadaću. Ako je potrebno, napišite dodatno pojašnjenje Vašeg rješenja.

1. Na dnu je dan primjer višenitnog programa implementiranog u programskom jeziku C, u kojem jedna nit izvodi funkciju *firstFunction*, a druga nit funkciju *secondFunction*. Prva funkcija povećava vrijednost varijable1, a smanjuje vrijednost varijable2, dok druga nit radi obrnuto. Ukoliko vrijednost varijable1 postane veća od vrijednosti varijable2, može se izvoditi samo druga nit. Također, ako vrijednost varijable2 postane veća od vrijednosti varijable1, može se izvoditi samo prva nit. Vaš zadatak je dovršiti programski kod tako da omogućite sinkronizaciju među nitima koristeći mutekse i uvjetne varijable iz biblioteke pthreads. Kopirajte programski kod i objasnite svoje rješenje. Kod kompajliranja dodajte opciju **-lm** zbog matematičke biblioteke.
2. U programskom kodu iz zadatka 1 zamijenite sinkronizaciju pomoću muteksa i uvjetnih varijabli sa sinkronizacijom pomoću semafora.
3. Proširite programski kod iz primjera 4 tako da implementirate sinkronizaciju problema pet filozofa koristeći monitore.

4.1. Zadaci za vježbu

Ove zadatke nije potrebno riješiti tijekom laboratorijske vježbe. Oni služe isključivo za vježbanje za pismeni ispit, a svoja rješenja možete provjeriti na konzultacijama.

1. Implementirajte sinkronizaciju problema pušača cigareta (Patil, 1971.) upotrebom semafora. Opis problema: Zamislite sustav s tri niti pušača i jednom niti trgovcem. Svaki pušač neprestano savija cigarete i puši. Kako bi se savila i popušila cigareta potrebno je imati tri sastojka: duhan, papir i šibice. Jedan pušač ima u neograničenim količinama samo papir, drugi samo duhan, a treći samo šibice. Trgovac ima sva tri sastojka u neograničenim količinama. Trgovac nasumice stavlja na stol dva različita sastojka. Pušač koji ima treći sastojak uzima sastojke sa stola, signalizira trgovcu, savija cigaretu i puši. Trgovac stavlja nova dva sastojka na stol i ciklus se ponavlja. Na početku je stol prazan. Napisati niti pušača i trgovca tako da se one međusobno ispravno sinkroniziraju s pomoću dva semafora. Napisati početne vrijednosti semafora.
2. Implementirajte sinkronizaciju problema proizvođač-potrošač pomoću muteksa i uvjetnih varijabli, ali pretpostavite da je moguće postojanje proizvoljnog broja potrošača i proizvođača u sustavu.
3. Implementirajte sinkronizaciju problema proizvođač-potrošač pomoću monitora, ali pretpostavite da je moguće postojanje proizvoljnog broja potrošača i proizvođača u sustavu.
4. Implementirajte sinkronizaciju problema starog mosta koristeći monitore. Stari most je uski most i stoga postavlja ograničenja na promet. Na njemu istovremeno smiju biti najviše tri automobila koja voze u istom smjeru. Simulirati automobile niti *Auto* koja obavlja radnje: *Popni_se_na_most (smjerA)* i *Siđi_s_mosta (smjerA)*. Napisati pseudokod monitorskih funkcija *Popni_se_na_most (smjer)* i *Siđi_s_mosta (smjer)*.

5. Zadan je sustav sa 4 resursa čiji je broj instanci definiran vektorom E = [6 5 8 7]. Pretpostavimo da je u memoriji u nekom trenutku 4 procesa (A, B, C, D) sa sljedećom tablicom alociranih resursa C te tablicom potrebnih resursa R.

$$\mathbf{C} = \begin{pmatrix} A & 1 & 1 & 1 & 1 \\ B & 2 & 0 & 1 & 3 \\ C & 2 & 3 & 4 & 1 \\ D & 0 & 0 & 2 & 0 \end{pmatrix} \qquad \mathbf{R} = \begin{pmatrix} A & 1 & 0 & 0 & 2 \\ B & 2 & 1 & 1 & 2 \\ C & 3 & 2 & 3 & 5 \\ D & 2 & 2 & 2 & 2 \end{pmatrix}$$

Potrebno je provjeriti mogu li se procesi izvršiti (je li sustav rješiv) ili će doći do zastoja. Koristite Bankarev algoritam. Zadatke riješite ručno, a korake rješenja skicirajte i opišite.

```
#include <stdio.h>
#include <pthread.h>
#include <math.h>

float variable1=10;
float variable2 = 9;

void *firstFunction() {
	while(1){
		//Radi samo ako je variable2>=variable1

		variable1 = 3*variable1*variable1 + 4*variable1 - 3;
		variable2 = sqrt(abs(variable2)) - 4;
		printf("Thread1: %.2f %.2f\n", variable1, variable2);

		//Obavijesti drugu nit ako je variable1>=variable2
	}
	pthread_exit(0);
}

void *secondFunction() {
	while(1){
		//Radi samo ako je variable1>=variable2

		variable1 = sqrt(abs(variable1))-4;
		variable2 = 3*variable2*variable2 + 4*variable2 - 3;
		printf("Thread2: %.2f %.2f\n", variable1, variable2);

		//Obavijesti prvu nit ako je variable2>=variable1
		}
	pthread_exit(0);
}
int main(void) {
	pthread_t thread1, thread2;
	pthread_create(&thread1, NULL, firstFunction, NULL);
	pthread_create(&thread2, NULL, secondFunction, NULL);
	pthread_join(thread1, NULL);
	pthread_join(thread2, NULL);
	return 0;
}
```

6. LITERATURA

[1] Stallings, W., 2012. Operating systems: internals and design principles. Boston: Prentice Hall,.

[2] Tanenbaum, A.S. and Bos, H., 2015. Modern operating systems. Pearson.

[3] Bovet, D.P. and Cesati, M., 2005. Understanding the Linux Kernel: from I/O ports to process management. " O'Reilly Media, Inc.".

[4] Love, R., 2010. Linux kernel development. Pearson Education.

[5] Love, R., 2013. Linux system programming: talking directly to the kernel and C library. " O'Reilly Media, Inc.".

[6] Jelenković L., 2019. Operacijski sustavi: Interni material za predavanja iz predmeta. Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva (http://www.zemris.fer.hr/~leonardo/os/fer/_OS-skripta.pdf)

[7] https://www.geeksforgeeks.org/operating-systems/

[8] http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

[9] https://www.baeldung.com/java-dining-philoshophers

[10] https://www.geeksforgeeks.org/dining-philosophers-solution-using-monitors/