Usporedba algoritama međusobnoga isključivanja

Dario, Bogović

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Dario Bogović

USPOREDBA ALGORITAMA MEĐUSOBNOGA ISKLJUČIVANJA

ZAVRŠNI RAD

Varaždin, 2018.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Dario Bogović

Matični broj: 43999/15–R

Studij: Informacijski sustavi

USPOREDBA ALGORITAMA MEĐUSOBNOGA ISKLJUČIVANJA

ZAVRŠNI RAD

Mentor:

Luka Milić, mag. ing. comp.

Varaždin, rujan 2018.

Dario Bogović

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

U višedretvenom načinu rada sve dretve koje djeluju unutar istog procesa dijele sva sredstva koje je operacijski sustav stavio na raspolaganje tom procesu. Prilikom pristupa nekom zajedničkom resursu važno je sinkronizirati dretve na način da u svakom trenutku samo jedna dretva ima pravo pristupa. Mehanizmi međusobnoga isključivanja osiguravaju dretvama naizmjeničan pristup dijeljenom resursu. U radu su opisani i implementirani neki od najpoznatijih algoritamskih rješenja problema međusobnoga isključivanja. Najveći nedostatak ovakvih rješenja je radno čekanje. Dretve koje čekaju na ulazak u kritični odsječak će neprestano ispitivati varijablu dok ona ne promijeni svoju vrijednost čime se troši vrijeme i resursi procesora. Umjesto njih, postoje ugradbeni mehanizmi koji rješavaju problem radnog čekanja i značajno smanjuju iskorištenost procesora čime ubrzavaju izvođenje operacijskog sustava. U ovome radu sam opisao neke od razreda i knjižnica pomoću kojih se u programskom jeziku C++ može na jednostavan način kontrolirati pristup nekom dijeljenom resursu. Provedena je usporedba brzine rada algoritama i ugradbenih mehanizama ovisno o operacijskom sustavu na kojem se izvode.

Ključne riječi: međusobno isključivanje; algoritmi; višezadaćnost; operacijski sustavi; dretve;

Sadržaj

1. Uvod

Računalo obavlja neki zadatak tako da izvodi programe pripremljene u višem programskom jeziku. Kada se program preveden u strojni oblik pokrene, on dobiva razna obilježja procesa, odnosno pripisuju mu se vremenska svojstva kao što su: trenutak početka izvođenja programa, trajanje izvođenja programa, trenutak završetka izvođenja programa, zaustavljanje izvođenja programa i slično. Odvijanje procesa obavlja se izvođenjem njegova niza instrukcija, tj. njegove dretve. Unutar procesa mora postojati barem jedna dretva. Proces se obavlja tako da procesor izvodi tu dretvu.

S obzirom na to da pojedini procesi u raznim fazama svojeg izvršavanja raznoliko troše pojedine dijelove računalnog sustava, pokazalo se razumnim posao organizirati tako da se u istom vremenskom razdoblju izvodi više zadataka. Dakle, u višeprocesorskim računalima proces možemo podijeliti na više dretava. Primjerice, ako jedan proces čeka na završetak ulazne operacije kako bi mogao nastaviti neko izračunavanje, za to vrijeme drugi proces može izvoditi svoje instrukcije. Takav višezadaćni rad omogućuje s jedne strane bolju porabu svih sredstava računalnog sustava, a s druge strane olakšava organizaciju poslova koje se obavljaju u računalnom sustavu. Stoga su u suvremenim operacijskim sustavima uvedeni mehanizmi koji podržavaju izvođenje procesa s više dretava pa govorimo o višedretvenom načinu rada ili višedretvenosti. U današnje vrijeme, svi računalni sustavi podržavaju višedretveni rad, iako se višedretveni rad može čak proizvesti i u jednoprocesorskom računalu tako da procesor naizmjenice „provlači“ jednu od više dretava. Dretve koje djeluju unutar jednog procesa dijele sva sredstva koja je operacijski sustav stavio na raspolaganje tom procesu. Posebice se to odnosi na adresni prostor procesa. Prema tome, sve dretve istog procesa mogu neposredno pristupiti do svih adresa adresnog prostora svog procesa. Međutim, dva procesa ne mogu jedan drugomu neposredno adresirati varijable. Ako je potrebno obaviti razmjenu podataka između dva procesa, onda se to mora činiti posredno s pomoću mehanizama koje osigurava operacijski sustav. (Budin, Golub, Jakobović, Jelenković, 2010.)

Ovaj rad opisuje neke od najpoznatijih rješenja međusobnoga isključivanja većeg broja dretava. U drugom poglavlju su navedeni razni mehanizmi koji se pritom mogu rabiti, dok se u trećem poglavlju detaljno razrađuju algoritamska rješenja problema. U programskim jezicima već postoje razni ugradbeni mehanizmi, stoga u četvrtom poglavlju su opisani oni koji se mogu rabiti u programskom jeziku C++. U petom poglavlju je obavljena usporedba između algoritamskih rješenja i ugradbenih mehanizama na pojedinom operacijskom sustavu.

2. Mehanizmi međusobnoga isključivanja

Mehanizam međusobnoga isključivanja osigurava da se neka sredstva računalnog sustava rabe pojedinačno. Dijelovi dretava koji rabe neko zajedničko sredstvo se zovu kritičnim odsječcima. Dretve, koje su inače nezavisne, smiju prolaziti kroz svoj kritični odsječak samo pojedinačno. Kada se dretve izvode u preostalom dijelu (nekritičnom odsječku) mogu se izvoditi proizvoljnom brzinom i redoslijedom. (Budin, Golub, Jakobović, Jelenković, 2010.)

Razlikujemo programsko i sklopovsko rješenje ostvarenja međusobnoga isključivanja dretava. Programska rješenja se postižu rabeći radno čekanje, a neka programska rješenja koja će biti i obrađena u ovom radu su:

- Dekkerov algoritam
- Petersonov algoritam
- Lamportov algoritam
- Eisenberg-McGuierov algoritam
- Szymanskijev algoritam
- Taubenfeldov algoritam

Danas postoje razni sinkronizacijski mehanizmi koji omogućavaju dretvama pristup nekom resursu računalnog sustava bez konflikata. Najpoznatiji sinkronizacijski mehanizmi su: monitori, uvjetne varijable, semafori, barijere i spinlockovi.

Sklopovska rješenja se postižu tako da ukoliko procesor već izvodi neku dretvu u kritičnom odsječku, on će držati svoj signal postavljenim u dva uzastopna nedjeljiva sabirnička ciklusa, a dodjeljivač sabirnice neće dodijeliti sabirnicu drugom procesoru dok se taj signal traženja ne poništi. Postoje različite vrste aktivnosti koje procesor može izvoditi u ta dva ciklusa:

- *„ispitati i postaviti“* (eng. „*test and set*“)
- *„zamijeniti“* (eng. „*swap*“)
- *„uvećaj i dodaj“* (eng. *„fetch-and-add“*)

3. Algoritmi međusobnoga isključivanja

3.1. Dekkerov algoritam

Prema WikiVisually, Dekkerov algoritam je prvo poznato ispravno rješenje problema međusobnoga isključivanja za dvije dretve. Rješenje se pripisuje nizozemskom matematičaru Theodorusu Jozefu Dekkeru. Pretpostavimo da postoje cikličke dretve koje se natječu za dijeljeni resurs. Tada kodni segment procesa možemo podijeliti na dva dijela: pristup zajedničkom resursu (kritični odsječak) i preostali dio (nekritični odsječak). U svakom trenutku najviše jedna dretva može biti u kritičnom odsječku, a ako je neka dretva već u kritičnom odsječku, druga dretva mora pričekati da prva izađe da bi on mogao ući u njega. Osnovna ideja za implementaciju Dekkerovog algoritma je da dretva odmah ulazi u kritični odsječak ako druga dretva ne zahtijeva ulazak u kritični odsječak. Ako su obadvije dretve istovremeno zainteresirane za pristup kritičnom odsječku, tada se propušta ona koji je najkasnije pristupila kritičnom odsječku.

Dekkerov algoritam je rješenje temeljeno na uporabi više varijabli. Svaka dretva koja pristupa dijeljenom resursu ima svoju vlastitu varijablu (zastavicu) koja pokazuje da li se zajednički resurs rabi ili ne. Zastavice su zajedničke i vidljive svim dretvama, ali svaka može postaviti samo svoju zastavicu, ne i tuđu. Osim zastavice, postoji i varijabla `PRAVO` koja se ispituje samo onda kada obje dretve istovremeno podignu svoje zastavice. Ako neka dretva želi ući u kritični odsječak kada je druga izvan kritičnog odsječka, onda se varijabla `PRAVO` uopće ne ispituje. Bez obzira na to, na izlasku iz kritičnog odsječka varijabla `PRAVO` se svaki put obnavlja tako da se pravo ulaska u kritični odsječak prepusti suprotstavljenoj dretvi, ako ona želi ući u svoj kritični odsječak. (Varga, 1994.)

Prema Budinu i sur. (2010), ovo je pseudokod za Dekkerov algoritam:

```
Dok je (1) {
      ZASTAVICA[I] = 1;
      Dok je (ZASTAVICA[J] != 0) {
            Ako je (PRAVO != I) {
                  ZASTAVICA[I] = 0;
                  Dok je (PRAVO != I);
                  ZASTAVICA[I] = 1;
            }
      }

      Kritični odsječak;

      PRAVO = J;
      ZASTAVICA[I] = 0;
      Nekritični odsječak;
}
```

Prema ovom algoritmu, dretva odmah ulazi u kritični odsječak kada druga dretva se ne natječe za ulazak u kritični odsječak. Ako pretpostavimo da su obadvije dretve istovremeno zainteresirane za ulazak u kritični odsječak, tada će obadvije istovremeno postaviti svoju zastavicu na 1 i ući u petlju, ali će ona dretva koji nema pravo morati privremeno odustati od zahtjeva za ulazak u kritični odsječak i čekati dok mu suprotstavljena dretva ne da pravo za ulazak u kritični odsječak. Nakon što dretva završi izvođenje kritičnog odsječka, prvo postavlja pravo drugoj dretvi i zatim spušta svoju zastavicu koja označava da se nalazi u kritičnom odsječku. Nakon toga, dretva može početi izvršavati svoj nekritični odsječak, a suprotstavljena dretva može slobodno ući u svoj kritični odsječak. (Alagarsamy, 2003.)

3.1.1. Poopćeni Dekkerov algoritam

Primjenu Dekkerovog algoritma za više dretava, opisao je K. Alagarsamy u svom radu „*Some Myths About Famous Mutual Exclusion Algorithms*“. Poopćenje Dekkerovog algoritma je napravio tako da je uveo logičko polje veličine `N` koje dretva postavlja kao istinito kada se nalazi u kritičnom odsječku. Osim toga, postoji i cjelobrojno polje u koji se zapisuje redoslijed pristupa kritičnom odsječku. Također su implementirane dvije funkcije. Jedna provjerava da li se neki drugi proces već nalazi u redu čekanja za kritični odsječak, a druga funkcija pomiče prioritete za ulazak u kritični odsječak. Prema Alagarsamyu, ovo je pseudokod za poopćeni Dekkerov algoritam:

```
Zauzet (K) {
      PRONAĐEN = 0;
      Za (J = 0 DO N-1) {
            Ako je (K!=J && ZASTAVICA[J] != 0)
                  PRONAĐEN = 1;
      }
      Vrati (PRONAĐEN);
}

Prilagodi () {
      Za (J = 0 DO N-1 && Red[J] != -1) {
            Red[J] = Red[J++];
      }
      Red[J] = -1;
}

PoopćeniDekkerovAlgoritam (I) {

      ZASTAVICA[I] = 1;
      Ako je (Zauzet(I) == 1) {
            ZASTAVICA[I] = 0;
            K = 0;
            Dok je (Red[K] != -1) K++;
            Red[K] = I;
            Dok je (Red[0] != I);
            ZASTAVICA[I] = 1;
      }
```

```
	Kritični odsječak;

	Prilagodi();
	ZASTAVICA[I] = 0;
	Nekritični odsječak;
}
```

Prema ovom algoritmu, dretva prvo postavlja svoju zastavicu i time daje znak drugim dretvama da ima namjeru ući u kritični odsječak. Zatim ispituje da li je neka druga dretva već postavila zastavicu, odnosno da li pokušava ući ili je već ušla u kritični odsječak. Ako ne postoji takva dretva, onda dretva može slobodno ući u kritični odsječak. Ako već postoji dretva koja se nalazi u kritičnom odsječku, tada dretva spušta zastavicu i čeka u redu čekanja dok ne dobije pravo, odnosno dok se ne izredaju sve druge dretve koji su prije nje zahtijevale ulazak. Kada ostvari pravo, dretva ponovno podiže svoju zastavicu i kreće s izvršavanjem kritičnog odsječka. Nakon što završi svoje izvršavanje, dretva spušta zastavicu i prilagođava red čekanja tako da pomiče druge dretve ispred, a sebe izbacuje iz reda. Proces potom slobodno izvršava nekritični odsječak.

Dekkerov algoritam sam implementirao u programskom jeziku C++:

```
#include <iostream>
#include <cstdlib>
#include "algoritmi.h"
#define N 5
using namespace std;

static int zastavica[N], red[N];

bool zauzet(int I) {
	for(int i = 0; i<N; i++) {
		if(i != I && zastavica[i] != 0) {
			return true;
		}
	}
	return false;
}

void dekkerUdiUKO(int I) {

	static bool initializedDekker;
	if(!initializedDekker) {
		fill_n(zastavica, sizeof(zastavica), 0);
		fill_n(red, sizeof(red), -1);

		initializedDekker = true;
	}

	zastavica[I] = 1;

	if (zauzet(I)) {
		zastavica[I] = 0;

		int K = 0;
		while(red[K] != -1) {
```

```
                K = K + 1;
            }
            red[K] = I;
            while(red[0] != I);

            zastavica[I] = 1;
      }
      else {
            red[0] = I;
      }
}

void dekkerIzidiIzKO(int I) {

      int K = 0;
      while(red[K] != -1 && K<(N-1)) {
            red[K] = red[K+1];
            K = K + 1;
      }
      red[K] = -1;
      zastavica[I] = 0;

}
```

3.2. Petersonov algoritam

Petersonov algoritam je još jedno rješenje koji omogućuje dretvama da rabe neko zajedničko sredstvo bez konflikata. Rješenje je implementirao američki znanstvenik Gary L. Peterson u svom radu „*Myths About the Mutual Exclusion*“ iz 1981. godine. Iako Petersonov originalni algoritam je primjenjiv za dva procesa, on se može poopćiti za više procesa.

Prema Budinu i sur. (2010), Petersonov algoritam nije ništa drugo nego malo pojednostavljenje Dekkerova algoritma. Varijabla `PRAVO` u Petersonovom algoritmu preimenuje se u `NEMA_PRAVO`. Svaka dretva koja želi ući u kritični odsječak postavlja svoju zastavicu i nakon toga zapisuje u varijablu `NEMA_PRAVO` svoj indeks. Ako obje dretve istovremeno žele ući u kritični odsječak, onda će varijabla `NEMA_PRAVO` poprimiti vrijednost indeksa one dretve čiji je procesor zadnji dobio pravo pristupa na sabirnicu. Nakon toga dretva čeka u petlji tako dugo dok suprotstavljena petlja ne spusti svoju zastavicu, odnosno dok ne obavi svoj kritični odsječak.

Ovako izgleda pseudokod Petersonovog algoritma za dva procesa koja se bore za pristup kritičnom odsječku:

```
Dok je (1) {

      ZASTAVICA[I] = 1;
      NEMA_PRAVO = I;
      Dok je ((NEMA_PRAVO == I) && (ZASTAVICA[J] == 1));
```

```
        Kritični odsječak;

        ZASTAVICA[I] = 0;
        Nekritični odsječak;

}
```

Dvije dretve imaju indekse 0 i 1, a početne vrijednosti varijabli `NEMA_PRAVO` i `ZASTAVICA` su 0. Ako samo jedna od dretava, recimo dretva 0, želi ući u kritični odsječak, ona će postaviti `NEMA_PRAVO` = 0 i `ZASTAVICA[0]` = 1. Nakon toga će ustanoviti da uvjet za ostanak u petlji nije ispunjen, jer `ZASTAVICA[1]` jednaka 0, i ući će odmah u kritični odsječak. Ako obje dretve istovremeno postave svoje zastavice, onda će u petlji čekalici ostati ona dretva koja je zadnja uspjela upisati svoj indeks u varijablu `NEMA_PRAVO` (čijem je procesoru dodjeljivač sabirnice drugom po redu dodijelio sabirnicu za pisanje u varijabli `PRAVO`). Suprotstavljena dretva će vidjeti da njezin uvjet za ponavljanje petlje nije ispunjen i ući će u svoj kritični odsječak. Tek kada spusti svoju zastavicu, uvjet za čekanje prestaje biti ispunjen i dretva ulazi u svoj kritični odsječak, dok će suprostavljena dretva svoje izvršavanje nastaviti u nekritičnom dijelu koda.

3.1.2. Poopćeni Petersonov algoritam

Petersonov algoritam koji sam prethodno opisao je primjenjiv za dvije dretve koja se bore za ulazak u kritični odsječak. Slično kao i kod Dekkera, algoritam možemo poopćiti tako da bude primjenjiv za više dretava. Umjesto varijable `NEMA_PRAVO` i polja zastavica, uvodi se cjelobrojno polje `RAZINA`, koje opisuje na kojoj se razini nalaze dretve na putu u kritični odsječak, i cjelobrojno polje `NEMA_PRAVO` koje opisuje koja dretva na pojedinoj razini nema pravo ulaska u kritični odsječak. Uvođenjem novih varijabli ovako izgleda pseudokod poopćenog Petersonovog algoritma:

```
Dok je (1) {

        Za (J = 1 DO N-1) {
                RAZINA[I] = J;
                NEMA_PRAVO[J] = I;
                Dok je ((∃K != I)(RAZINA[K] >= J && NEMA_PRAVO[J] == I));
        }

        Kritični odsječak;
        RAZINA[I] = 0;

        Nekritični odsječak;

}
```

Postoji N−1 razina kroz koje dretva mora proći kako bi ušla u kritični odsječak. Algoritam će dopustiti da najviše N dretava bude u prvoj razini, N−1 dretava u drugoj razini, …, tri dretve u razini N−2 i dvije dretve u posljednjoj (N−1) razini, tako da samo jedna dretva na temelju ostvarenog prava može nastaviti izvršavati kritični odsječak. Pri ulasku u novu razinu, dretva postavlja svoj indeks polja na broj razine u kojoj se trenutno nalazi, postavlja svoj indeks u polje NEMA_PRAVO na toj razini i čeka u petlji čekalici sve dok dretve koje se nalaze na višoj razini ne završe izvršavanje kritičnog odsječka. Napredovanje kroz razine se može postići i ako neka druga dretva na istoj razini postavi svoj indeks u polju NEMA_PRAVO za tu razinu. To se događa kada se više dretava nalazi na istoj razini, pa tada prolaze dalje sve dretve osim one koja nema pravo prolaska dalje, odnosno one dretve koja je zadnja ušla u razinu. (Herlihy i Shavitm 2008.)

U najboljem slučaju, ako samo jedna dretva zahtijeva ulazak u kritični odsječak ona će postaviti razinu na kojoj se nalazi i sebe postaviti da nema pravo prolaska u više razine. Iako je postavila sebe da nema pravo, ipak neće čekati u petlji pošto ne postoje dretve na višim razinama dretva uspješno prolazi sve preostale razine i ulazi u kritični odsječak.

U najgorem slučaju, ako istovremeno sve dretve zahtijevaju ulazak u kritični odsječak, u višu razinu će proći sve dretve osim one koja je zadnja postavila sebe da nema pravo prolaska dalje. Za primjer ćemo pretpostaviti da istovremeno postoji pet dretava koje žele ući u kritični odsječak. One će morati proći kroz četiri razine ili četiri vrata kako bi došle do cilja. U prvom ciklusu će četiri dretve proći u drugu razinu, a jedna, koja je zadnja postavila polje NEMA_PRAVO na indeksu razine će čekati u petlji. U drugom ciklusu dalje prolazi tri dretve u treću razinu, a jedna će čekati u petlji. Slično će u trećem ciklusu dvije proći u posljednju četvrtu razinu, a na kraju ona najbrža dretva ulazi u kritični odsječak. Dretve koje su zastale u petljama će čekati na prolazak dalje sve dok suprotstavljene dretve iz viših razina ne završe svoje izvršavanje u kritičnim odsječcima.

Petersonov algoritam sam implementirao u programskom jeziku C++:

```
#include <iostream>
#include <cstdlib>
#include "algoritmi.h"
#define N 5
using namespace std;

static int razina[N], nemaPravo[N];

void petersonUdiUKO(int I) {
      static bool initializedPeterson;
      if(!initializedPeterson) {
            fill_n(razina, sizeof(razina), 0);
            fill_n(nemaPravo, sizeof(nemaPravo), 0);
            initializedPeterson = true;
      }
```

```
	for(int J = 1; J<N; J++) {

		razina[I] = J;
		nemaPravo[J] = I;

		bool cekaj;
		do {
			cekaj = false;
			for(int K = 0; K<N; K++) {
				if(razina[K] >= J && nemaPravo[J] == I && K != I)
cekaj = true;
			}
		}
		while(cekaj);
	}
}

void petersonIzidiIzKO(int I) {

	razina[I] = 0;

}
```

3.3. Lamportov algoritam

Mnoga ranija rješenja međusobnoga isključivanja većeg broja dretava su se pokazala kompliciranom i teško razumljivom zbog velikog broja zajedničkih varijabli i složenih petlji koje određuju ulazak u kritični dio. Lamportov algoritam je prvo ispravno rješenje međusobnoga isključivanja većeg broja dretava. Autor rješenja, Leslie Lamport, ga je prozvao još i pekarskim algoritmom. Naime, zamisao postupka se može objasniti na primjeru pekare u kojoj jedan prodavač poslužuje kupce. Prodavač može poslužiti najviše jednog kupca istovremeno. Kada je jedan kupac prisutan, stvar je jednostavna: kupac uđe u pekaru, zahtijeva kruh, prodavač ga posluži i kupac napušta trgovinu. Problem nastaje kada više kupaca istovremeno zahtijeva prodavača. Kako se kupci ne bi gurali oko pulta iza kojeg stoji prodavač, na ulazu u prodavaonicu se dijele brojevi. Svi kupci vide broj kupca koji se upravo poslužuje, pa pultu bez dodatnog guranja pristupa kupac s sljedećim brojem. Kupci bivaju posluženi onim redom kojim su ušli u pekaru. (Deitel, Deitel, Choffnes, 2004.)

Kupci koji žele kupiti kruh su zapravo dretve koje žele ući u kritični odsječak. U strukturi podataka svaka dretva će dobiti svoju varijablu u kojoj zapisuje broj dobiven „pri ulasku u trgovinu“. Te varijable su svrstane u poredak `BROJ[N]`. Osim toga, svaka dretva dobiva još jednu nadzornu varijablu koja označava da se dretva nalazi u fazi dodjele broja i da stoga treba pričekati s ispitivanjem dodijeljenog broja. Te dodatne nadzorne varijable svih dretava se mogu svrstati u poredak `ULAZ[N]`. Prema tome, ovako izgleda pseudokod Lamportovog algoritma:

```
Dok je (1) {

      ULAZ[I] = 1;
      BROJ[I] = max(BROJ[0], …, BROJ[n-1]) + 1;
      ULAZ[I] = 0;


      Za (J = 0 do N) {
            Dok je (ULAZ[J] == 1);
            Dok je ((BROJ[J] != 0) && (BROJ[J] < BROJ[I] || (BROJ[J] ==
BROJ[I] && J < I));
      }

      Kritični odsječak;

      BROJ[I] = 0;
      Nekritični odsječak;
}
```

Dretva kada želi ući u kritični odsječak prvo postavlja zastavicu `ULAZ`, prema kojoj drugim dretvama daje do znanja da se nalazi u postupku uzimanja broja. Uzet će broj za jedan veći od najvećeg broja koji ima neka suprotstavljena dretva. U takvom postupku dodjele brojeva mogu nastati problemi. Naime, ako dvije ili više dretava istovremeno žele ući u kritični odsječak, sve one mogu u uzastopnim sabirničkim ciklusima pročitati istu staru vrijednost zadnjeg dodijeljenog broja prije nego li se taj broj uveća. Ta se pojava u radnom okruženju u kojem se dretve izvode ne može nikako izbjeći.

Moguće rješenje proizlazi iz zamisli Dekkerovog algoritma. Tamo se postupalo tako da se od dvije dretve koje su postavile zastavice, i time najavile želju za ulazak u kritični odsječak, odabrala ona na koju je pokazivala varijabla `PRAVO`. Ovdje se može dogovoriti da vrijednost indeksa određuje pravo ulaska onda kada dretve dobiju jednake brojeve. To znači da je početnim dodjeljivanjem indeksa dretvama unaprijed utvrđen redoslijed ulaska u kritični odsječak. Nakon što je dretva „ušla kroz vrata“ i dodijeljen joj je njezin broj, ona započinje pregledavanje svih ostalih dretava.

U petlji, dretva može zastati na dva mjesta: ispitujući varijablu `ULAZ[J]`, ako suprotstavljena dretva upravo traži dodjelu broja ili dok dretva koja ima prednost (`BROJ[J]` manji od `BROJ[I]`) ne izađe iz kritičnog odsječka i zapiše `BROJ[J]`. Prvi kratki zastanak potreban je da se izbjegne čitanje varijable `BROJ[J]` u razdoblju kada se on upravo mijenja. U drugoj petlji čekalici dretva će ostati tako dugo dok dretva čiji je broj bio manji ne izvrši svoj kritični odsječak do kraja i postavi u svoju varijablu `BROJ` vrijednost 0. Isto tako, ako postoje dretve koje imaju jednake brojeve, dretva s većim indeksom će pričekati dok dretve s manjim indeksom završe izvršavanje svog kritičnog odsječka. (Budin, Golub, Jakobović, Jelenković, 2010.)

Lamportov algoritam sam implementirao u programskom jeziku C++:

```
#include <iostream>
#include <cstdlib>
#include "algoritmi.h"
#define N 5
using namespace std;

static int ulaz[N], broj[N];

void lamportUdiUKO(int I) {
	static bool initializedLamport;
	if(!initializedLamport) {
		fill_n(ulaz, sizeof(ulaz), 0);
		fill_n(broj, sizeof(broj), 0);

		initializedLamport = true;
	}

	int J;

	ulaz[I] = 1;
	for (J=0; J<N; J++) if (broj[J]>broj[I]) broj[I] = broj[J];
	broj[I] = broj[I] + 1;
	ulaz[I] = 0;

	for (J=0; J<N; J++)  {
		while (ulaz[J] != 0);
		while (broj[J] !=0 && (broj[J]<broj[I] || (broj[J]==broj[I] &&
J < I)));
	}

}

void lamportIzidiIzKO(int I) {
	broj[I] = 0;
}
```

3.4. Eisenberg-McGuireov algoritam

Prije predstavljanja Lamportovog algoritma, brojni računalni znanstvenici su tražili načine kako omogućiti međusobno isključivanje većeg broja dretava. Tako su 1972. godine Murray A. Eisenberg i Michael R. McGuire predstavili svoj algoritam međusobnoga isključivanja. Prema Taubenfeldu (2008), iako algoritam ispravno radi jer se u svakom trenutku samo jedna dretva nalazi u kritičnom odsječku, on sadrži i nekoliko nedostataka. Algoritam se sastoji od nekoliko jednosmjernih skokova na određenu liniju koda (bezuvjetni skokovi) što smanjuje preglednost i razumljivost koda. Također, dretve neće ulaziti u kritičnim odsječak FIFO redoslijedom, već je teoretski moguće da neka dretva koja je kasnije zatražila zahtjev za ulazak u kritični odsječak uđe prije dretve koja je zahtjev zatražila prije. Zbog toga, Eisenberg-McGuireov algoritam, iako ispravno radi, nije najbolji primjer međusobnoga isključivanja dretava.

Algoritam se sastoji od dvije zajedničke varijable: red zastavica i još jedne zajedničke varijable `K`. Zastavice mogu poprimiti vrijednosti 0, 1 ili 2 gdje 0 označava da dretva trenutno nije zainteresirana za ulazak u kritični odsječak, 1 označava da je dretva zainteresirana za ulazak u kritični odsječak i 2 koji pokazuje da dretva ima prednost ulaska u kritični odsječak. Postoji i varijabla `K` koju vide sve dretve i ona označava indeks dretve koja sljedeća ima prednost ulaska u kritični odsječak. Postavlja ju dretva nakon izvršetka svog kritičnog odsječka. (Lynch, Goldman, 1989.)

Pseudokod za Eisenberg-McGuireov algoritam je prikazan u nastavku:

```
Dok je (1) {

      Start:
      ZASTAVICA[I] = 1;

      Ponavljaj {
            J = K;
            Dok je (ZASTAVICA[J] == 0) J = 1 + (J % N);
      }
      Dok je (J != I);

      ZASTAVICA[I] = 2;
      Za (J = 0 do N) {
            Ako je (J != I && ZASTAVICA[J] == 2) idi na Start;
      }

      Ako je (ZASTAVICA[K] != 0 && K != I) idi na Start;
      K = I;

      Kritični odsječak;

      J = K;
      Ponavljaj {
            J = 1 + (J%N);
            Ako je (ZASTAVICA[J] != 0) K = J;
      }
      Dok je (ZASTAVICA[J] == 0);
      ZASTAVICA[I] = 0;

      Nekritični odsječak;

}
```

Prema Eisenberg-McGuirevom algoritmu, kada dretva zatraži ulazak u kritični odsječak prvo će postaviti svoju zastavicu na vrijednost 1. U svakom trenutku u varijabli `K` je zapisan indeks dretve koja ima prednost. Dretva koja ima prednost će bez problema proći sve daljnje sigurnosne petlje i ući u kritični odsječak, a ona koja nema prednost mora pričekati dok joj suprotstavljena dretva nakon izvršavanja kritičnog odsječka ne dodijeli prednost. Obično, prednost dobiva sljedeća dretva s većim indeksom, a koja ima postavljenu zastavicu za ulazak u kritični odsječak. Ako dretva koja ima prednost ne zatraži ulazak u kritični odsječak, dretva s sljedećim većim indeksom ima pravo ulaska u kritični odsječak.

Dretve prolaze dvije sigurnosne provjere prije ulaska u kritični odsječak. U prvoj provjeri obavlja se filtriranje dretava ako ih je više od jedne istovremeno prošlo prvu while petlju. Prednost će uvijek imati ona dretva s manjim indeksom koju je prethodno predodredio procesor, a sve ostale dretve moraju ponovno na početak. U zadnjoj sigurnosnoj provjeri promatra se je li dretva koja ima prednost u međuvremenu zatražila ulazak u kritični odsječak. Ako nije, dretva može slobodno sebe označiti da ima prednost i početi izvršavat kritični odsječak. Ali ako je u međuvremenu dretva koja ima prednost zatražila ulazak u kritični odsječak, dretva će morati propustiti nju da izvrši do kraja kritični odsječak. Nakon izvršavanja kritičnog odsječka, dretva određuje tko sljedeći ima prednost, poništava svoju zastavicu i dalje nastavlja izvršavati nekritični odsječak.

Eisenberg-McGuierov algoritam sam implementirao u programskom jeziku C++:

```
#include <iostream>
#include <cstdlib>
#include "algoritmi.h"
#define N 5
using namespace std;

static int zastavica[N];

void eisenbergMcGuireUdiUKO(int I) {

	static bool initializedEisenbergMcGuire;
	if(!initializedEisenbergMcGuire) {
		fill_n(zastavica, sizeof(zastavica), 0);

		initializedEisenbergMcGuire = true;
	}

	int J, K = 0;

	start:
	zastavica[I] = 1;

	do {
		J = K;
		while(zastavica[J] == 0) J = 1 + (J%N);
	}
	while(J != I);

	zastavica[I] = 2;
	for(J = 0; J<N; J++) {
		if(J != I && zastavica[J] == 2) goto start;
	}

	if (zastavica[K] != 0 && K != I) goto start;
	K = I;

}

void eisenbergMcGuireIzidiIzKO(int I) {

	zastavica[I] = 0;
}
```

3.5. Szymanskijev algoritam

Szymanskijev algoritam je algoritam međusobnoga isključivanja kojeg je implementirao računalni znanstvenik Boleslaw Szymanski. Rješenje je predstavio 1988. godine u svom radu *„A Simple Solution to Lamport's Concurrent Programming Problem with Linear Wait“*. Ideja algoritma je jednostavna. U početku sve dretve koje namjeravaju ući u kritični odsječak postavljaju zastavicu i čekaju u čekaonici. Kada nema više dretava koje zahtijevaju ulaz, dretve jedna po jedna izlaze iz čekaonice i izvršavaju kritični odsječak. Ako u tom trenutku se pojave dretve koje zahtijevaju ulaz u kritični odsječak, one će pričekati ispred čekaonice dok prethodne dretve koje se nalaze u čekaonici ne izvrše kritični odsječak do kraja.

U čekaonici gdje se dretve nalaze postoje dva vrata: jedna za ulaz, a druga za izlaz iz čekaonice. Na početku su vrata za ulaz otvorena, a vrata za izlaz zatvorena. Svaka dretva pri ulasku u čekaonicu provjerava da li neka druga dretva trenutno ima namjeru ući u kritični odsječak. Ako takva dretva postoji, vrata za ulaz ostaju otvorena i dretva ulazi u čekaonicu. Inače, zatvaraju se ulazna vrata u čekaonici i otvaraju se vrata za izlaz iz čekaonice, te dretva izvršava kritični odsječak. Ako postoji više dretava u čekaonici, u kritični odsječak ulazi ona s najmanjim indeksom koje je prethodno predodredio procesor.

Dretve dijele jednu varijablu (zastavicu) koja opisuje u kojem se stanju trenutno nalaze. Postoji pet stanja u kojima se dretva može nalaziti:

- 0 – označava da se dretva trenutno nalazi u nekritičnom odsječku
- 1 – ukazuje da dretva ima namjeru ući u kritični odsječak
- 2 – označava da dretva čeka dok preostale dretve koje su ukazale da imaju namjeru ući u kritični odsječak uđu u čekaonicu
- 3 – označava da se dretva nalazi u čekaonici
- 4 – ukazuje da je dretva izašla iz čekaonice i čeka na izvršavanje kritičnog odsječka

Prema ovim brojevima stanja u kojima se pojedina dretva može nalaziti, ovako izgleda pseudokod Szymanskijevog algoritma:

```
Dok je (1) {

      ZASTAVICA[I] = 1;

      Dok je (∀J≠I, ZASTAVICA[J] >= 3);
      ZASTAVICA[I] = 3;

      Za (J = 0 DO N-1) {
            Ako je (ZASTAVICA[J] == 1) ZASTAVICA[I] = 2;
      }

      Dok je (∃J≠I, ZASTAVICA[J] != 4);
```

```
        ZASTAVICA[I] = 4;
        Dok je (∀J<I, ZASTAVICA[J] >= 2);

        Kritični odsječak;
        Dok je (∀J>I, ZASTAVICA[J] == 2 || ZASTAVICA[J] == 3);

        ZASTAVICA[I] = 0;
        Nekritični odsječak;

}
```

Primjećujemo kako se algoritam sastoji od četiri dijela:

1. Prvi dio u kojem dretva čeka iza vrata za ulazak u čekaonicu. Čekanje se vrši dok sve suprotstavljene dretve izađu iz čekaonice i do kraja izvrše svoj kritični odsječak
2. Drugi dio u kojem se otvaraju ulazna vrata u čekaonicu i gdje dretva čeka na zatvaranje ulaznih vrata i otvaranje izlaznih vrata čekaonice, odnosno dok sve dretve koje imaju namjeru za ulazak uđu u čekaonicu
3. Treći dio u kojem čeka na izvršenje kritičnog odsječka. Prednost imaju dretve s manjim indeksom.
4. Posljednji, četvrti dio, u kojem dretva izvršava kritični odsječak, te na kraju sprječava da se ne zatvore vrata za izlaz iz čekaonice za one dretve koje čekaju na izvršavanje kritičnog odsječka. Szymanski navodi da će algoritam biti ispravan i ako se to čekanje premjesti liniju ispod postavljanja `ZASTAVICA[I]` = 4;

Za primjer ćemo pretpostaviti da postoje tri dretve koje se bore za ulazak u kritični odsječak. Svaka će postaviti svoju zastavicu na stanje 1 čime ukazuju da imaju namjeru ući u kritični odsječak. Nakon toga svaka ispituje da li postoje neke dretve koje se nalaze u čekaonici ili izvršavaju kritični odsječak. Trenutno takve dretve ne postoje, stoga dretve izlaze iz petlje čekalice i nastavljaju dalje s izvršavanjem algoritma tako da daju znak da su ušle u čekaonicu. Zatim one „brže“ dretve čekaju u petlji čekalici dok sporije dretve uđu u čekaonicu. Zadnja dretva koja je iskazala namjeru ulaska, zatvara vrata ulaza i postavljanjem svoje zastavice u stanje 4 otvara vrata izlaza iz čekaonice. Prednost ulaska u kritični odsječak imaju dretve s manjim indeksom. Nakon izvršavanja kritičnog odsječka slijedi sigurnosna provjera da li su sve dretve koje se nalaze u čekaonici postavile svoju zastavicu u stanje 4. Nakon toga, dretva postavlja svoju zastavicu u stanje 0 pri čemu označava da se trenutno nalazi u nekritičnom odsječku. Ako u trenutku dok se tri dretve nalaze u fazi izvršavanja kritičnog odsječka, stvore nove dvije dretve koje žele ući u kritični odsječak, one će morati pričekati u petljama da prve tri dretve izvrše do kraja svoj kritični odsječak i vrate se na stanje s brojem 0.

Szymanskijev algoritam sam implementirao u programskom jeziku C++:

```
#include <iostream>
#include <cstdlib>
#include "algoritmi.h"
#define N 5
using namespace std;

static int zastavica[N];

bool otvoriUlaz(int I){
	for(int K = 0; K<N; K++) {
		if(K != I && zastavica[K] >= 3) return true;
	}
	return false;
}
bool zatvoriUlaz(int I){
	for(int K = 0; K<N; K++) {
		if(K != I && zastavica[K] != 4) return false;
	}
	return true;
}
bool kriticniOdsjecak(int I) {
	for(int K = 0; K<I; K++) {
		if(zastavica[K] >= 2) return true;
	}
	return false;
}
bool ostaviVrataOtvorena(int I) {
	for(int K = I+1; K<N; K++) {
		if(zastavica[K] == 2 || zastavica[K] == 3) return true;
	}
	return false;
}

void szymanskiUdiUKO(int I) {
	static bool initializedSzymanski;
	if(!initializedSzymanski) {
		fill_n(zastavica, sizeof(zastavica), 0);

		initializedSzymanski = true;
	}

	zastavica[I] = 1;

	while(otvoriUlaz(I));
	zastavica[I] = 3;

	for(int J = 0; J<N; J++) {
		if(zastavica[J] == 1) zastavica[I] = 2;
	}
	while(zatvoriUlaz(I));

	zastavica[I] = 4;
	while(kriticniOdsjecak(I));
}

void szymanskiIzidiIzKO(int I) {
	while(ostaviVrataOtvorena(I));
	zastavica[I] = 0;
}
```

3.6. Taubenfeldov algoritam

Taubenfeldov algoritam (eng. *The Black-White Bakery Algorithm*) je još jedan algoritam međusobnoga isključivanja koji je malo poboljšanje Lamportovog algoritma. U radu u kojem je predstavio svoj algoritam, Taubenfeld je zadao tri važna svojstva koje algoritam međusobnoga isključivanja mora imati: dretve moraju ulaziti u kritični odsječak onim redom kojim zahtijevaju ulazak, moraju biti prilagodljive svakom problemu i rabiti konačnu veličinu memorije registra. Kao nedostatak Lamportovog algoritma naveo je to što je moguće da neka dretva uđe u kritični odsječak prije nego neka druga dretva koja u petlji čekalici više čeka, te kako brojevi koje dretve uzimaju rastu u beskonačnost što zauzima veliku količinu podataka u registrima. Na temelju tih problema Taubenfeld je u svom radu „*The Black-White Bakery Algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms*„ predstavio svoj algoritam koji rješava probleme Lamportovog algoritma.

U odnosu na Lamportov algoritam, Taubenfeldov algoritam se sastoji od još jedne zajedničke varijable `BOJA` koja može poprimiti vrijednost 0 (bijela) ili 1 (crna). Naime, kupac koji ulazi u pekaru će uz broj dobiti i boju. Broj više neće rasti u beskonačnost već do ukupnog broja dretava čime se postiže smanjenje zauzeća memorije u odnosu na Lamportov algoritam. Prema tome, dretva će pri zahtijevanju za ulazak u kritični odsječak pročitati vrijednost zajedničke varijable `BOJA` i postaviti ga u svoju varijablu `MOJA_BOJA`. Zatim će uzeti broj koji je za jedan veći od broja karte koje imaju dretve s istom bojom i čeka u petlji čekalici dok njegov broj ne bude najmanji, odnosno dok ne dođe na red. U slučaju da dvije dretve koje imaju isti broj, a sadrže različite boje, prednost će imati ona dretva koja ima boju različitu od boje zapisane u varijabli `BOJA`. U slučaju da dvije dretve koje imaju iste brojeve imaju i iste boje, tada presuđuje indeks dretve pri čemu prednost ima dretva s manjim indeksom. Pri izlasku iz kritičnog odsječka, dretva će postaviti varijablu `BOJA` u suprotnu boju od one koju je dretva imala. Pseudokod Taubenfeldovog algoritma je prikazan u nastavku:

```
Dok je (1) {

      ULAZ[I] = 1;
      MOJA_BOJA[I] = BOJA;

      Za (J = 0 do N) {
            Ako je (MOJA_BOJA[J] > MOJA_BOJA[I]) BROJ[I] = BROJ[J];
      }

      BROJ[I] = BROJ[I] + 1;
      Ako je (BROJ[I] == N) BROJ[I] = 1;


      ULAZ[I] = 0;
```

```
      Za (J = 0 do N) {
            Dok je (ULAZ[J] == 1);
            Ako je (MOJA_BOJA[I] == MOJA_BOJA[J]) {
                  Dok je (BROJ[J] != 0 && (BROJ[J] < BROJ[I] || (BROJ[J] ==
BROJ[I] && J < I)) && MOJA_BOJA[I] == MOJA_BOJA[J]);
            }
            Inače {
                  Dok je (BROJ[J] != 0 && MOJA_BOJA[I] == BOJA && MOJA_BOJA[I]
!= MOJA_BOJA[J]);
            }
      }
      Kritični odsječak;

      Ako je (MOJA_BOJA[I] == CRNA) {
            BOJA = BIJELA;
      }
      Inače {
            BOJA = CRNA;
      }
      BROJ[I] = 0;

      Nekritični odsječak;

}
```

Pretpostavimo da postoje tri dretve koje se bore za ulazak u kritični odsječak. Na početku će svaka uzeti boju. Na primjer, pretpostavimo da je na početku postavljena bijela boja, pa će svaka imati bijelu boju. Dretve zatim ispituju koji je trenutni najveći broj koji ima neka suprotstavljena dretva s istom bojom. Pošto ne postoje dretve koje se nalaze u kritičnom odsječku, najveći broj je trenutno nula. Taj broj se povećava za jedan i time dretva dobiva svoj broj te postavlja zastavicu `ULAZ` na 0 čime daje znak kako je završila s „ulazom u pekaru“. Dretva čeka u petlji čekalici sve dok dretve iz prvog ciklusa ne dobiju svoje brojeve.

Zatim se ispituju redom dretve od onih s manjim do onih s većim indeksom s obzirom da se preskaču one koje nisu uzele brojeve, odnosno nisu zatražile ulazak u kritični odsječak. Ako dretve sadrže istu boju, čekanje se vrši sve dok dretve s većim prioritetom (brojem ili indeksom dretve) ne završe izvršavanje svog kritičnog odsječka. Ako dretve sadrže različitu boju, one čekaju sve dok se ne promijeni zastavica `BOJA` u vrijednost suprotnu od vrijednosti svoje boje koje ima dretva. Kada ostvari prednost, dretva ulazi i izvršava kritični odsječak, postavlja zastavicu `BOJA` u suprotnu od onog koju ima čime daje prednost dretvama sa svojom bojom, te poništava svoj broj i izvršava dalje nekritični odsječak. Pretpostavimo da se u trenutku dok dretve čekaju na ulazak u kritični odsječak, stvore nove dvije dretve. Obadvije će dobiti crnu boju, jer su prethodne dretve pri izlasku iz kritičnog odsječka postavile varijablu `BOJA` na crno. One će u petlji morati pričekati dok sve dretve s bijelom bojom ne izvrše kritični odsječak.

Primjećujemo kako je cijeli smisao Taubenfeldovog algoritma utemeljen na tome da dretve koje puno kasnije zatraže ulazak u kritični odsječak, ne uđu prije onih koje su prije

zatražile što je teoretski bilo moguće kod Lamportovog algoritma. Taubenfeldov algoritam sam implementirao u programskom jeziku C++:

```
#include <iostream>
#include <cstdlib>
#include "algoritmi.h"
#define N 5
#define BIJELA 0
#define CRNA 1
using namespace std;

static int ulaz[N], broj[N], mojaBoja[N];
static int boja = BIJELA;

void taubenfeldUdiUKO(int I) {
      static bool initializedTaubenfeld;
      if(!initializedTaubenfeld) {
            fill_n(ulaz, sizeof(ulaz), 0);
            fill_n(broj, sizeof(broj), 0);
            fill_n(mojaBoja, sizeof(mojaBoja), 0);

            initializedTaubenfeld = true;
      }

      int J;

      ulaz[I] = 1;
      mojaBoja[I] = boja;

      for (J=0; J<N; J++) if (broj[J]>broj[I]) broj[I] = broj[J];
      broj[I] = broj[I] + 1;
      if (broj[I] == N) broj[I] = 1;

      ulaz[I] = 0;

      for (J=0; J<N; J++)  {
              while (ulaz[J] != 0);
              if(mojaBoja[I] == mojaBoja[J]) {
                  while(broj[J]  !=  0  &&  (broj[J]<broj[I]  ||  (broj[J]  ==
broj[I] && J < I)) && mojaBoja[I] == mojaBoja[J]);
            }
            else {
                  while(broj[J] != 0 && mojaBoja[I] == boja && mojaBoja[I]
!= mojaBoja[J]);
            }
      }
}

void taubenfeldIzidiIzKO(int I) {

      if(mojaBoja[I] == CRNA) boja = BIJELA;
      else boja = CRNA;
      broj[I] = 0;

}
```

4. Ugradbeni mehanizmi međusobnoga isključivanja

U prethodnom poglavlju su objašnjeni neki od prvih rješenja međusobnoga isključivanja većeg broja dretava. Glavni nedostatak takvih algoritamskih rješenja je radno čekanje. Procesor neprestano ispituje varijablu dok ona ne promijeni svoju vrijednost čime se nepotrebno troši vrijeme i resursi procesora, ali i sabirnički ciklusi koji mogu biti iskorišteni na puno bolji način. Umjesto toga, osmišljeni su razni programski sinkronizacijski mehanizmi koji rješavaju problem radnog čekanja i značajno smanjuju iskorištenost procesora čime ubrzavaju izvođenje operacijskog sustava. Najpoznatiji sinkronizacijski mehanizmi su: monitori, uvjetne varijable, semafori, barijere i spinlockovi.

4.1. POSIX Threads

POSIX Threads ili `pthreads` su izvršni modeli koji omogućavaju kontrolu pristupa nekom dijeljenom resursu. Implementacija je dostupna na svim operacijskim sustavima sličnim Unixu poput Linux, Mac OS X, Android, Solaris i FreeBSD. Na Microsoft Windowsu je također dostupan zahvaljujući paketu `pthreads-w32` koji implementira `pthreads`.

Pthreads definira skupinu tipova podataka, funkcija i konstantni namijenjenom programskom jeziku C. Sadržan je u knjižnici `pthreads.h`. Postoji oko 100 dostupnih naredbi, sve sadrže prefiks `pthreads_`, i organizirane su u četiri grupe: upravljači dretava (kreiranje dretava i sl.), binarni semafori (eng. mutexes), uvjetne varijable (eng. condition variables) i sinkronizacija dretava.

U pthreads se za zaštitu dijeljenih podataka i implementaciju monitora rabe binarni semafori (eng. mutexes). Postoje dva moguća stanja koja može imati *mutex*: zaključan (biti pod vlasništvom neke dretve) ili otključan (biti slobodan). Naredbe za *mutexe* imaju prefiks `pthreads_mutex_` i opisane su u nastavku:

- `int` **`pthread_mutex_init`** `(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);` - dinamički se inicijalizira novi *mutex* objekt
- **`pthread_mutex_t`** – statički se inicijalizira novi *mutex* objekt. Prilikom inicijalizacije je moguće pridružiti sljedeće vrijednosti koje određuju vrstu *mutexa*: `PTHREAD_MUTEX_INITIALIZER` (brzi *mutexi*), `PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP` (rekurzivni *mutexi*), i `PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP` (*mutexi* koje provjeravaju greške).

- `int` **`pthread_mutex_lock`** `(pthread_mutex_t *mutex));` - zaključava *mutex* kojeg primi u argumentu. Ako je *mutex* trenutno otključan, on postaje zaključan i jedino ga dretva koja ga je zaključala može otključat. Ako je *mutex* trenutno zaključan, dretva će čekati dok se ne otključa (brzi *mutex*), javiti grešku `EDEADLK` (*mutexi* koji ispituju greške) ili izbrojiti koliko je puta dretva zaključala *mutex*, tako da kad bi se *mutex* poslije želio otključati mora se izvršiti taj broj pthread_mutex_unlock naredbi (rekurzivni *mutexi*)
- `int` **`pthread_mutex_trylock`** `(pthread_mutex_t *mutex);` - ponaša se identično kao i pthread_mutex_lock, ali ne blokira dretvu ako je *mutex* već zaključan već vraća grešku „EBUSY“
- `int` **`pthread_mutex_unlock`** `(pthread_mutex_t *mutex);` - otključava *mutex* (brzi *mutex*) ili smanjuje broj potrebnih pthread_mutex_unlock operacija za otključavanje *mutexa* (rekurzivni *mutex*). *Mutex* se otključava ako je broj takvih potrebnih operacija nula.
- **`int pthread_mutex_destroy`** `(pthread_mutex_t *mutex);` - uništava *mutex* objekt oslobađajući resurse koje je eventualno zauzeo. Prilikom uništavanja *mutex* mora biti otključan. U Linuxu *mutex* objekt ne zauzima resurse, pa se ovom funkcijom provjerava da li je *mutex* otključan.

Funkcija `pthread_mutex_init` uvijek vraća 0, dok ostale metode vraćaju 0 ako je izvršenje uspješno ili neki drugi broj ako se dogodila greška.

(Unix man pages)

4.2. Razred std::Mutex

U programskom jeziku C++ za zaštitu dijeljenih podataka se može rabiti razred Mutex definirana u knjižnici Mutex. Metode u razredu omogućuju zaštitu dijela koda, sprječavajući suprotstavljenim dretvama pristup istim memorijskim lokacijama. Za razliku od `pthreads`, razred Mutex ne podržava rekurzivne *mutexe*, odnosno dretva ne može zaključati *mutex* koji je već zaključan. Zaključavanje i otključavanje *mutexa* se provodi pomoću tri metode opisane u nastavku:

- **`lock`** – metoda pomoću koje se zaključava *mutex*
- **`try_lock`** – metoda koja će pokušati zaključati *mutex* i vratiti vrijednost koja označava da li je *mutex* uspješno zaključan
- **`unlock`** – metoda pomoću koje se otključava *mutex*

Dretva pozivom funkcije `lock` ili `try_lock` traži pristup kritičnom odsječku. Ako već postoji dretva koja se nalazi u kritičnom odsječku, tada će sve druge dretve koje zahtijevaju pristup biti blokirane (pozivom funkcije `lock`) ili vratiti `false` vrijednost (pozivom funkcije `try_lock`). Dretva posjeduje *mutex* do onda kada ga otključa pozivom funkcije `unlock`.

(cppreference.com)

4.3. Knjižnica <threads.h>

U programskom jeziku C je, uz `pthreads`, dostupna još jedna knjižnica koja omogućuje međusobno isključivanje dretava i nalazi se u zaglavlju `threads.h`. Naredbe za *mutexe* imaju prefiks `mtx_` i opisane su u nastavku:

- **`mtx_t`** – identifikator *mutexa*
- `int` **`mtx_init`** `( mtx_t* mutex, int type );` - funkcija koja kreira novi *mutex* objekt. Prvi parametar je pokazivač na *mutex* koji se želi inicijalizirati, a drugi parametar predstavlja tip *mutexa*. Tip može biti: `mtx_plain` (jednostavni *mutex*), `mtx_timed` (*mutex* koji podržava vremenski odmak), `mtx_plain | mtx_recursive` (rekurzivni *mutex*) i `mtx_timed | mtx_recursive` (rekurzivni *mutex* s vremenskim odmakom)
- `int` **`mtx_lock`** `( mtx_t* mutex );` - zaključava se *mutex* koji se primi u argumentu. Ako je *mutex* već zaključan, dretva čeka dok ga suprotstavljena dretva ne otključa.
- `int` **`mtx_timedlock`** `( mtx_t *restrict mutex, const struct timespec *restrict time_point );` - kod *mutexa* koji podržavaju vremenski odmak, dretva zaključava *mutex* definiran u prvom argumentu koji će biti zaključan sve dok ga dretva ne otključa ili dok ne prođe vrijeme zadano u drugom argumentu.
- `int` **`mtx_trylock`** `( mtx_t *mutex );` - zaključava *mutex* ukoliko on nije zaključan, odnosno ako je zaključan vratiti će vrijednost thrd_busy.
- `int` **`mtx_unlock`** `( mtx_t *mutex );` - otključava *mutex* kojeg primi u argumentu.
- `void` **`mtx_destroy`** `( mtx_t *mutex );` - uništava *mutex* objekt oslobađajući zauzetu memoriju.

Sve funkcije, osim `mtx_destroy`, vraćaju `thrd_success` ako je funkcija ispravno izvršena ili `thrd_error` ako se dogodila greška. Funkcija `mtx_trylock` će uz to vratiti i `thrd_busy` ako je *mutex* već zaključan. (cppreference.com)

4.4. Knjižnica <windows.h>

Na operacijskom sustavu Microsoft Windows, u programskom jeziku C, postoji skupina *mutex* objekata koje omogućavaju dretvama pristup dijeljenom resursu. Navedena skupina *mutex* objekata se nalazi u zaglavlju `windows.h`.

- **`CreateMutex`** – funkcija koja prima 4 argumenta i pomoću koje se kreira novi *mutex*. Prvi argument je uvijek `NULL`. Drugi argument definira sigurnost *mutexa*, a ako je on naveden kao `NULL`, *mutex* će imati osnovna svojstva sigurnosti. Treći argument definira vlasnika *mutexa*; ako je vrijednost `TRUE`, onaj proces koji ga je kreirao ima vlasništvo nad *mutexom*, inače *mutex* nema vlasništvo. Četvrti argument predstavlja naziv *mutexa*, te ako je on naveden kao `NULL` onda je *mutex* neimenovan.
- **`WaitForSingleObject`** – funkcija koju dretva poziva kada zahtjeva ulazak u kritični odsječak. Prvi argument funkcije predstavlja *mutex*, dok drugi argument određuje koliko je maksimalno vrijeme čekanja na ulazak u kritični odsječak. Dretva čeka na ulazak dok suprotstavljena dretva ne oslobodi *mutex* ili dok ne istječe vrijeme čekanja na ulazak.
- **`OpenMutex`** – funkcija pomoću koje se dretva može služiti s imenovanim *mutexima*, tako da u argument funkcije navede naziv *mutexa* s kojim se želi služiti.
- **`ReleaseMutex`** – funkcija kojom se oslobađa zauzeti *mutex* čime se daje pravo drugim dretvama da ulaze u kritični odsječak. Argument funkcije je *mutex*. Vrlo je važno da se nakon izlaska iz kritičnog odsječka pozove funkcija `ReleaseMutex` jer može doći do zastoja ili program može nepravilno raditi.

Sve funkcije vraćaju vrijednost `NULL` ako se dogodila greška u izvršavanju, a vrijednost koja je različita od `NULL` predstavlja uspješno izvršenje funkcije.

(Microsoft Docs)

4.5. Razred System::Threading::Mutex

Microsoft je u .NET Frameworku razvio razred Mutex pomoću koje dretve mogu nesmetano pristupiti nekom zajedničkom resursu. Razred se nalazi u okolini `System.Threading`., a naredbe su opisane u nastavku:

- **Mutex**(Boolean, String, Boolean, MutexSecurity) – konstruktor razreda Mutex. Prvi argument je tipa `Boolean` i njime se definira početno vlasništvo dretve nad *mutexom*. Drugi argument je tipa `String` i njime se definira naziv *mutexa*. Treći argument je tipa `Boolean` i on također služi za definiranje vlasništva nad *mutexom*. Zadnji argument konstruktora definira sigurnost pristupa *mutexu*. Važno je napomenuti da je *mutex* moguće kreirati i bez navedenih argumenata.
- **WaitOne**() – metoda kojom dretva zahtijeva ulazak u kritični odsječak. Ako je zauzet, dretva čeka dok se ne oslobodi. Kao argument metode se može zadati i vremenski period koji označava koliko najdulje dretva čeka na ulazak u kritični odsječak.
- **ReleaseMutex**() – metoda kojom dretva izlazi iz kritičnog odsječka i otključava *mutex*
- **OpenExisting**(String) – dretva ulazi u kritični odsječak pomoću *mutexa* navedenog u argumentu
- **TryOpenExisting**(String) – dretva pokušava ući u kritični odsječak pomoću *mutexa* navedenog u argumentu. Metoda vraća vrijednost koja govori da li je operacija uspješno izvedena
- **SetAccessControl**(MutexSecurity) – definiranje sigurnosti pristupa *mutexu*
- **Equals**(Object), **ToString**(), **GetType**() … - metode karakteristične za .NET Framework

U razredu `System::Threading::Mutex` moguće je kreirati dva tipa *mutexa*: imenovani i neimenovani. Imenovanim *mutexima* se u svakom trenutku može pristupiti pomoću metoda `OpenExisting` ili `TryOpenExisting`. Za razliku od imenovanih koji su vidljivi u cijelom operacijskom sustavu, neimenovani *mutexi* su vidljivi samo dretvama koje imaju referencu na *mutex* objekt. Metodom `WaitOne` dretva zahtijeva vlasništvo nad *mutexom* i čeka na ulazak u kritični odsječak dok *mutex* ne signalizira da je slobodan ili dok ne prođe unaprijed zadano vrijeme koliko dretva može najviše čekati. Kada završi s izvršavanjem kritičnog odsječka, obavezno se poziva metoda `ReleaseMutex` nakon kojeg dretva više nema vlasništvo nad *mutexom*. Rekurzivni *mutex* je podržan u ovom razredu na način da dretva može više puta pozvati `WaitOne` metodu, ali kako bi oslobodila *mutex* mora pozvati jednako toliko puta i metodu `ReleaseMutex`.

(Microsoft Developer Network, Mutex Class)

4.6. Razred CMutex

U Visual C++ postoji razred CMutex koja omogućuje kreiranje *mutexa*, sinkronizacijskih objekata koji se rabe pri međusobnom isključivanju većeg broja dretava. Razred se nalazi u zaglavlju `afxmt.h`. Konstruktor razreda se sastoji od tri argumenta. Prvi argument je tipa `Boolean` i on određuje da li dretva koja kreira *mutex* ima na početku pristup resursu kojeg kontrolira *mutex*. Ukoliko se on ne postavi, argument će poprimiti vrijednost false. Drugi argument predstavlja naziv *mutexa*, te ako se on ne navede ili poprimi vrijednost `NULL` *mutex* će biti neimenovan. Treći argument definira svojstva sigurnosti *mutexa*. Nakon izrade CMutex objekta, preporuča se da se preko metode `GetLastError` provjeri da li *mutex* već postoji.

Da bi pristupili ili oslobodili *mutex* potrebno je kreirati novi `CMultiLock` ili `CSingleLock` objekt, te pozvati njihove funkcije `Lock` ili `Unlock`. Prilikom kreiranja objekta, obavezno je u konstruktoru navesti pokazivač na prethodno kreirani *mutex* objekt. `CSingleLock` se rabi kada dretva može čekati na najviše jednom objektu istovremeno, dok `CMultiLock` se rabi ako dretva može čekati na više objekata istovremeno. Metode za zaključavanje i oslobađanje *mutexa* su opisane u nastavku:

- **`IsLocked`**() - provjerava da li je *mutex* objekt zaključan. Metoda vraća vrijednost koja je različita od nule ako je *mutex* objekt zaključan, a inače vraća vrijednost 0.
- **`Lock`**() - metoda pomoću koje dretva zahtijeva ulazak u kritični odsječak. Kao argument se može zadati maksimalno vrijeme čekanja u milisekundama. U slučaju da je *mutex* zaključan, odnosno ako već postoji suprotstavljena dretva koja se nalazi u kritičnom odsječku, dretva čeka da se *mutex* oslobodi. Metoda će vratiti grešku u slučaju ako prođe maksimalno vrijeme čekanja zadano u argumentu. Kada suprotstavljena dretva oslobodi *mutex*, dretva koja je čekala zaključava *mutex* i krene s izvršavanjem kritičnog odsječka.
- **`Unlock`**() - metoda kojom dretva izlazi iz kritičnog odsječka, otključava *mutex* i nastavlja s izvršavanjem nekritičnog odsječka.

(Microsoft Developer Network, CMutex Class)

5. Usporedba programskih rješenja međusobnoga isključivanja

U prethodnim poglavljima sam opisao najpoznatija algoritamska rješenja problema međusobnoga isključivanja i ugradbene mehanizme operacijskoga sustava na kojem se izvode. Svaki od opisanih rješenja je ispravan jer se pomoću njih omogućava dretvama da bez konflikata u svakom trenutku mogu pristupiti dijeljenom resursu. Iako rade istu stvar, razlikuju se po mnogim stvarima poput veličine memorije koje zauzimaju ili broju potrebnih instrukcija za obavljanje međusobnog isključivanja dretava. Svi ugradbeni mehanizmi se baziraju na inicijalizaciji *mutex* objekata, te njihovom zaključavanju i otključavanju. Postavlja se pitanje koje je od ovih programskih rješenja najefikasnije, odnosno u kojem će dretve najbrže ulaziti i izlaziti iz kritičnog odsječka.

Efikasnost programskih rješenja ću ispitati pomoću jednostavnog programa. Za svako rješenje zasebno ću stvoriti novu dretvu koja će određeni broj puta ulaziti i izlaziti iz kritičnog odsječka. Dretva će mjeriti potrebno vrijeme izvršavanja. Zatim, glavna dretva za svako rješenje će izračunati prosjek i ispisati vrijeme potrebno za ulazak i izlazak iz kritičnog odsječka. Program će se izvršiti na operacijskim sustavima Linux Ubuntu 18.04 i Microsoft Windows 10. Algoritmi koji će biti testirani su: Dekkerov algoritam, Petersonov algoritam, Lamportov algoritam, Eisenberg-McGuierov algoritam, Szymanskijev algoritam i Taubenfeldov algoritam. Uz algoritme, na Linuxu će se testirati ugradbeni mehanizmi iz razreda `std::Mutex` i POSIX Mutex, dok će se na Windowsu testirati ugradbeni mehanizmi iz razreda `System::Threading::Mutex` i knjižnice `windows.h`. Za mjerenje vremena rabit će se funkcija `timespec_get`. Na početku izvršavanja programa zadaje se broj željenih ulazaka i izlazaka iz kritičnog odsječka. Ispitat će se vrijeme koje je potrebno dretvama za ulazak i izlazak. Za primjer, ispitat će se prosječno vrijeme za 10, 100 ili 1000 ulazaka u kritični odsječak. U tablici 1, tablici 2 i tablici 3 su prikazana prosječna vremena ulaska i izlaska dretava iz kritičnog odsječka na operacijskom sustavu Linux, dok su u tablici 4, tablici 5 i tablici 6 prikazana prosječna vremena ulaska i izlaska dretava iz kritičnog odsječka na operacijskom sustavu Windows.

Tablica 1: Rezultat ispitnog programa u operacijskom sustavu Linux za 10 ulazaka

PROGRAMSKO RJEŠENJE	PROSJEČNO VRIJEME ULASKA	PROSJEČNO VRIJEME IZLASKA
Dekkerov algoritam	341 ns	260 ns
Petersonov algoritam	444 ns	221 ns
Lamportov algoritam	431 ns	217 ns
Eisenberg-McGuierov algoritam	408 ns	201 ns
Szymanskijev algoritam	535 ns	532 ns
Taubenfeldov algoritam	482 ns	491 ns
POSIX Mutex	882 ns	794 ns
Razred std::Mutex	938 ns	1116 ns

Tablica 2: Rezultat ispitnog programa u operacijskom sustavu Linux za 100 ulazaka

PROGRAMSKO RJEŠENJE	PROSJEČNO VRIJEME ULASKA	PROSJEČNO VRIJEME IZLASKA
Dekkerov algoritam	264 ns	315 ns
Petersonov algoritam	476 ns	352 ns
Lamportov algoritam	511 ns	350 ns
Eisenberg-McGuierov algoritam	348 ns	326 ns
Szymanskijev algoritam	558 ns	587 ns
Taubenfeldov algoritam	414 ns	530 ns
POSIX Mutex	957 ns	744 ns
Razred std::Mutex	744 ns	1043 ns

Tablica 3: Rezultat ispitnog programa u operacijskom sustavu Linux za 1000 ulazaka

PROGRAMSKO RJEŠENJE	PROSJEČNO VRIJEME ULASKA	PROSJEČNO VRIJEME IZLASKA
Dekkerov algoritam	233 ns	426 ns
Petersonov algoritam	459 ns	301 ns
Lamportov algoritam	410 ns	356 ns
Eisenberg-McGuierov algoritam	328 ns	360 ns
Szymanskijev algoritam	515 ns	636 ns
Taubenfeldov algoritam	415 ns	443 ns
POSIX Mutex	1049 ns	913 ns
Razred std::Mutex	1037 ns	1165 ns

Tablica 4: Rezultat ispitnog programa u operacijskom sustavu Windows za 10 ulazaka

PROGRAMSKO RJEŠENJE	PROSJEČNO VRIJEME ULASKA	PROSJEČNO VRIJEME IZLASKA
Dekkerov algoritam	10 080 ns	12 280 ns
Petersonov algoritam	9 480 ns	11 340 ns
Lamportov algoritam	10 070 ns	11 060 ns
Eisenberg-McGuierov algoritam	12 640 ns	11 150 ns
Szymanskijev algoritam	12 090 ns	11 800 ns
Taubenfeldov algoritam	12 070 ns	12 180 ns
Knjižnica <windows.h>	74 430 ns	63 060 ns
Razred System::Threading::Mutex	175 820 ns	318 860 ns

Tablica 5: Rezultat ispitnog programa u operacijskom sustavu Windows za 100 ulazaka

PROGRAMSKO RJEŠENJE	PROSJEČNO VRIJEME ULASKA	PROSJEČNO VRIJEME IZLASKA
Dekkerov algoritam	10 985 ns	16 974 ns
Petersonov algoritam	11 777 ns	14 151 ns
Lamportov algoritam	9 966 ns	13 343 ns
Eisenberg-McGuierov algoritam	15 210 ns	22 860 ns
Szymanskijev algoritam	15 313 ns	22 192 ns
Taubenfeldov algoritam	14 560 ns	26 413 ns
Knjižnica <windows.h>	39 919 ns	66 436 ns
Razred System::Threading::Mutex	65 162 ns	67 313 ns

Tablica 6: Rezultat ispitnog programa u operacijskom sustavu Windows za 1000 ulazaka

PROGRAMSKO RJEŠENJE	PROSJEČNO VRIJEME ULASKA	PROSJEČNO VRIJEME IZLASKA
Dekkerov algoritam	19 825 ns	27 442 ns
Petersonov algoritam	15 550 ns	16 619 ns
Lamportov algoritam	18 477 ns	24 161 ns
Eisenberg-McGuierov algoritam	17 862 ns	24392 ns
Szymanskijev algoritam	16 993 ns	24 253 ns
Taubenfeldov algoritam	16 424 ns	22 428 ns
Knjižnica <windows.h>	27 395 ns	33 670 ns
Razred System::Threading::Mutex	47 376 ns	46 574 ns

Primjećujemo da kod ovih primjera se u svim slučajevima algoritmi međusobnoga isključivanja zbog svoje jednostavnosti brže izvršavaju od ugradbenih mehanizama operacijskog sustava. Na operacijskom sustavu Windows prilikom povećanja broja ulazaka u kritični odsječak, prosječno vrijeme za ulazak i izlazak iz kritičnog odsječka kod ugradbenih

mehanizama se smanjuje, dok se kod algoritama povećava. Na operacijskom sustavu Linux povećanje broja ulazaka u kritični odsječak ne utječe drastično na prosječno vrijeme ulaska ili izlaska.

Ulazak u kritični odsječak je nešto brži u poopćenom Dekkerovom, poopćenom Petersonovom i Lamportovom algoritmu. Dekkerov algoritam za ulazak u kritični odsječak postavlja zastavicu i provjerava da li je neka druga dretva podigla zastavicu. Kako u ovom primjeru ne postoji suprotstavljena dretva u kritičnom odsječku, dretva će u njega brzo ući. Slično je i kod Petersonovog algoritma gdje će dretva slobodno prolaziti kroz razine i bez problema ući u kritični odsječak. U Lamportovom algoritmu dretvi će biti dodijeljen broj 1 pa će dretva brzo prolaziti kroz prvu petlju čekalicu, jer ne postoji niti jedna dretva koja je u fazi dodijele broja, i drugu petlju čekalicu jer je dodijeljeni broj najveći od svih brojeva. Najsporije se izvršava ulazak kod Szymanskijevog i Taubenfeldovog algoritma. U Szymanskijevom algoritmu dretva će prolaziti kroz niz provjera da li postoji suprotstavljena dretva koja je postavila neku vrijednost. U Taubenfeldovom algoritmu se uz broj dodjeljuje i boja, pa se za ulazak u kritični odsječak provjerava, osim broja, i boja koju dretva posjeduje. Ulazak u Eisenberg-McGuierovom algoritmu se na operacijskom sustavu Windows izvršava znatno dulje zbog velikog broja bezuvjetnih skokova koje algoritam sadrži.

Prosječno vrijeme izlaska iz kritičnog odsječka je najbrže kod algoritama kod kojih se izlazak vrši na način da dretva jednostavno spusti zastavicu. Prema tome, najbrže izlaze dretve koje rabe Petersonov algoritam, Lamportov algoritam i Eisenberg-McGuierov algoritam. Najdulji izlazak se odvija kod Dekkerovog algoritma zbog toga što pri izlasku dretva prilagođava red čekanja za ulazak u kritični odsječak tako što sebe izbacuje iz reda, a druge dretve pomiče ispred. Dretve u Taubenfeldovom algoritmu moraju još postaviti trenutnu boju i poništiti svoj broj, a kod Szymanskija dretva mora čekati dok sve suprotstavljene dretve koje su ukazale namjeru ući u kritični odsječak ne uđu u čekaonicu za ulazak.

Može se primjetiti kako su ovaj ispit najbrže prošle dretve koje su rabile Petersonov i Lamportov algoritam. Algoritmi su prilično jednostavni, za njihovu primjenu su dovoljna tek dva polja u kojoj će dretve postavljati svoje vrijednosti. Ulazak u kritični odsječak se obavlja uz mali broj instrukcija: kod Petersona se promatra ima li dretva pravo prolaska, dok kod Lamporta prednost prolaska uvijek imaju dretve koje imaju dodijeljeni manji broj. Izlazak iz kritičnog odsječka je još jednostavniji: dretve samo spuste zastavicu i mogu dalje izvršavati nekritični odsječak. S druge strane, Szymanskijev i Taubenfeldov algoritam su se pokazali kao najsporiji na ovome ispitu. Kod Szymanskijevog algoritma, dretve da bi ušle i izašle iz kritičnog odsječka moraju proći niz petlja čekalica, te postaviti svoju zastavicu na nekoliko vrijednosti. Taubenfeld se čini kao kompliciranija verzija Lamporta koja rabi još jednu dodatnu varijablu. Kada se na ovaj način promatra, Szymanskijev i Taubenfeldov algoritam se mogu činiti kao neefikasni

načini rješenja problema međusobnoga isključivanja dretava. Njihova prednost se očituje u tome što će dretve uvijek ulaziti u kritični odsječak onim redom kojim su zahtijevale ulazak. Naime, kod Petersonovog ili Lamportovog algoritma je vrlo lako moguće da dretve koje kraće čekaju prije uđu u kritični odsječak od onih koje dulje čekaju. Prema tome su Szymanskijev i Taubenfeldov algoritam puno pravedniji od svojih starijih prethodnika.

Primjećujemo kako se algoritamska rješenja međusobnoga isključivanja puno brži izvršavaju od ugradbenih mehanizama na operacijskim sustavima. Ipak, danas se pri izgradnji programskih proizvoda rabe uglavnom već postojeći ugradbeni mehanizmi. Glavni nedostatak algoritamskih rješenja je radno čekanje. Dretve koje čekaju u petljama čekalicama će neprestano ispitivati zastavicu i tako nepotrebno trošiti računalne resurse koji mogu biti iskorišteni na bolji način. Unatoč tome što su sporiji, ugradbeni mehanizmi blokiraju dretve koje čekaju na ulazak, pa je time trošenje resursa računala minimalno. U programskom jeziku C++, na operacijskom sustavu Linux je ispitana brzina ulaska i izlaska iz kritičnog odsječka, odnosno zaključavanja i otključavanja *mutexa* koji se nalaze u razredu `std::Mutex` i knjižnici `pthread.h`. Može se primjetiti da se otključavanje i zaključavanje najbrže odvija s *mutexima* iz knjižnice `pthread.h`, odnosno onih koji sadrže prefiks `pthread_`. Na operacijskom sustavu Windows je ispitano zaključavanje i otključavanje *mutexa* iz knjižnice `windows.h` i razreda `System::Threading::Mutex`. Ispitni program je pokazao da se znatno brže odvija zaključavanje i otključavanje *mutexa* iz knjižnice `windows.h`. Prema rezultatima ispitnog programa, iako svi ugradbeni mehanizmi rade istu stvar i niti jedan se ne izdvaja po većim mogućnostima nad *mutexima* koje ima u odnosu prema drugim ugradbenim mehanizmima, u programskom jeziku C++ i operacijskom sustavu Linux efikasnije je rabiti POSIX Mutex, dok je na operacijskom sustavu Windows efikasnije rabiti *mutexe* iz knjižnice `windows.h`.

6. Zaključak

U višedretvenom načinu rada sve dretve koje djeluju unutar istog procesa dijele sva sredstva koje je operacijski sustav stavio na raspolaganje tom procesu. Prilikom pristupa nekom zajedničkom resursu važno je sinkronizirati dretve na način da u svakom trenutku samo jedna dretva ima pravo pristupa. Programski kod je potrebno podijeliti na njegov kritični dio (u kojem dretva pristupa zajedničkom resursu) i nekritični dio. U svakom trenutku najviše jedna dretva mora biti u kritičnom dijelu, a ako već postoji dretva u kritičnom dijelu sve druge dretve čekaju ispred dok se kritični dio ne oslobodi. Mehanizmi međusobnoga isključivanja osiguravaju dretvama naizmjeničan pristup dijeljenom resursu. Razlikujemo programske i sklopovske mehanizme međusobnoga isključivanja.

U ovome radu su obrađeni neki od najpoznatijih algoritamskih rješenja međusobnoga isključivanja poput: Dekkerovog algoritma koji je prvo poznato ispravno rješenje, Petersonov algoritam koji je malo pojednostavljenje Dekkerovog algoritma, Lamportov algoritam koji je prvo ispravno rješenje međusobnoga isključivanja većeg broja dretava, Eisenberg-McGuierov algoritam koji je rješenje s bezuvjetnim skokovima, te novija rješenja poput Szymanskijevog i Taubenfeldovog algoritma. Algoritamska rješenja su ispravna, te se pomoću njih može upravljati dretvama na način da u svakom trenutku samo jedna od njih izvršava kritični dio koda. Najveći nedostatak ovakvih rješenja je radno čekanje. Dretve koje čekaju na ulazak u kritični odsječak će neprestano ispitivati varijablu dok ona ne promijeni svoju vrijednost čime se troši vrijeme i resursi procesora.

Umjesto njih, osmišljeni su razni mehanizmi koji rješavaju problem radnog čekanja i značajno smanjuju iskorištenost procesora čime ubrzavaju izvođenje operacijskog sustava. U ovome radu sam opisao neke od razreda i knjižnica pomoću kojih se u programskom jeziku C++ može na jednostavan način kontrolirati pristup nekom dijeljenom resursu. Svi ugradbeni mehanizmi se baziraju na inicijalizaciji *mutex* objekata, te njihovom zaključavanju i otključavanju. Ispitnim programom je ustanovljeno kako je u programskom jeziku C++ najefikasnije u operacijskim sustavima sličnim Unixu rabiti POSIX Mutex, dok u Microsoft Windowsu *mutex* objekt iz knjižnice `windows.h`.

Za kraj možemo zaključiti kako su mehanizmi međusobnoga isključivanja vrlo važne komponente operacijskog sustava. Bez njih se nikada ne bi mogao ostvariti ovakav višezadaćni način rada kakav imamo danas u suvremenom svijetu.

Popis literature

[1] Alagarsamy, K. (2003): Some Myths About Famous Mutual Exclusion Algorithms

[2] Budin, L., Golub, M., Jakobović, D., Jelenković, L. (2010): Operacijski sustavi

[3] Cppreference.com, *std::mutex*. Preuzeto 14.08.2018. s https://en.cppreference.com/w/cpp/thread/mutex

[4] Cppreference.com, *Thread support library*. Preuzeto 14.08.2018. s https://en.cppreference.com/w/c/thread

[5] Deitel, H.M., Deitel, P.J., Choffnes, D.R. (2004): Operating Systems

[6] Herlihy, M., Shavit, N. (2008): The Art of Multiprocessor Programming

[7] Lynch, N.A., Goldman, K.J. (1989): Distributed Algorithms

[8] Microsoft Developer Network, *CMutex Class.* Preuzeto 23.08.2018. s https://msdn.microsoft.com/en-us/library/tt45160e.aspx

[9] Microsoft Developer Network, *Mutex Class*. Preuzeto 14.08.2018. s https://msdn.microsoft.com/en-us/library/system.threading.mutex(v=vs.110).aspx

[10] Microsoft Docs, *Using Mutex Object.* Preuzeto 14.08.2018. s https://docs.microsoft.com/en-us/windows/desktop/sync/using-mutex-objects

[11] Szymanski, B., (1988): A Simple Solution to Lamport's Concurrent Programming Problem with Linear Wait

[12] Taubenfeld, G., (2004): The Black-White Bakery Algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms

[13] Taubenfeld, G., (2008): Synchronization Algorithms and Concurrent Programming

[14] Unix man pages, *PTHREAD_MUTEX.* Preuzeto 14.08.2018. s http://www.skrenta.com/rt/man/pthread_mutex_init.3.html

[15] Varga, M. (1994): Problematika međusobnog isključenja procesa implementirana različitim tehnikama

[16] WikiVisually, *Dekker's Algorithm.* Preuzeto 08.07.2018. s https://wikivisually.com/wiki/Dekker%27s_algorithm

Popis tablica