Sveučilište J. J. Strossmayera u Osijeku

Fakultet elektrotehnike, računarstva i informacijskih tehnologija

Kneza Trpimira 2b

HR-31000 Osijek

www.ferit.unios.hr

Laboratorijska vježba 7:

Upravljanje nitima i njihovo međusobno isključivanje unutar operacijskih sustava Linux i Windows

Sadržaj

1. UVOD

Koncept procesa unutar operacijskih sustava se sažeto može promatrati kroz njegove dvije osnovne karakteristike:

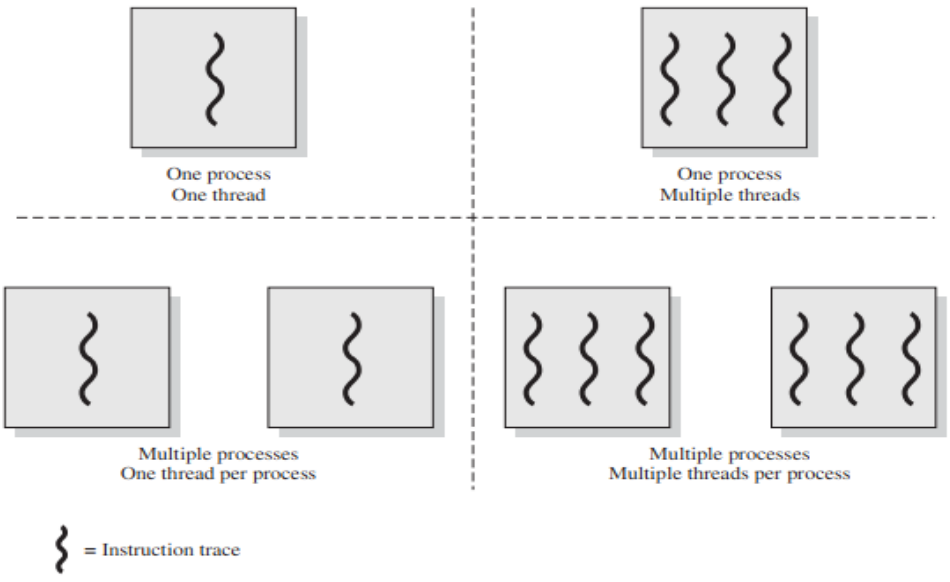
- Vlasništvo nad resursima – proces sadrži virtualni adresni prostor za spremanje slike procesa. Operacijski sustav također dodjeljuje procesima glavnu memoriju, I/O kanale i uređaje, i datoteke.
- Raspoređivanje/Izvršavanje – izvršavanje procesa slijedi putanju kroz jedan ili više programa, a može biti i prekinuto od strane drugih procesa. Operacijski sustav također upravlja stanjima procesa, prioritetom izvršavanja i raspoređivanja.

Kako bi se razlikovale ove dvije karakteristike procesa, jedinica izvršavanja još naziva laki proces ili **nit** (engl. *thread*), a jedinica vlasništva resursa proces ili **zadatak** (engl. *task*). Pojam **višenitnosti** (engl. *multithreading*) se odnosi na sposobnost operacijskog sustava da podržava višestruke, konkurentne putanje izvršavanja unutar istog procesa.

U ovoj laboratorijskoj vježbi pojasnit ćemo razloge razvijanja višenitnih aplikacija i pokazati način njihove izrade pomoću korisničkih niti. Objasnit ćemo probleme kontrole istovremenosti koji se javljaju u višenitnim aplikacijama te sinkronizacijske mehanizme koji rješavaju takve probleme.

1.1. Višenitnost

Tradicionalni pristup izvršavanja jedne niti po procesu se naziva jednonitni pristup. Razlika u jednonitnim i višenitnim pristupima je prikazana na Slici 1, tako da su u gornjoj polovici prikazani pristupi s jednim procesom izvršavanja, a dolje pristupi koje nude operacijski sustavi koji podržavaju izvođenje više programa istovremeno (*multiprogramming*). MS-DOS je primjer operacijskog sustava koji podržava samo jedan proces i jednu nit, dok UNIX podržava više procesa ali samo jednu nit po procesu. Desna strana slike se odnosi na višenitne pristupe. Primjerice, JRE (engl. *Java Runtime Environment*) u jednom trenutku podržava samo jedan proces s više niti. Moderni operacijski sustavi, kao Windows, Solaris i Linux podržavaju i višeprocesne i višenitne koncepte.



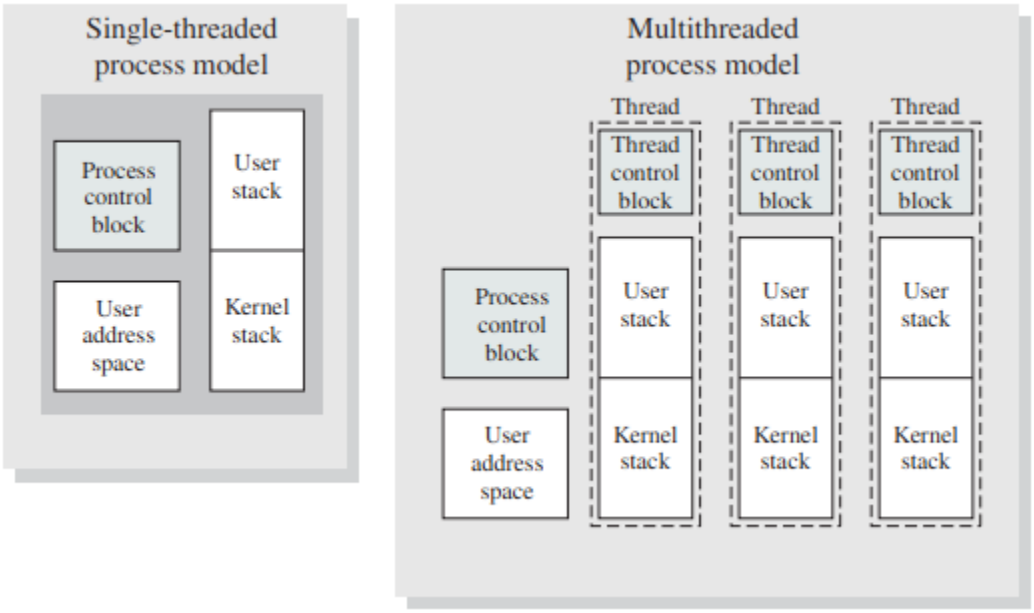
Slika 1. Razlika između koncepata jednonitnosti i višenitnosti

U višenitnom okruženju, proces se definira kao jedinica alokacije resursa i jedinica zaštite. Svakom procesu se dodjeljuje virtualni adresni prostor za pohranu slike procesa te zaštićeni pristup procesoru i ostalim procesima. Unutar samog procesa, postoji jedna ili više niti koje sadrže:

- Stanje izvršavanja niti (*Running, Ready, …*)
- Kontekst niti
- Stog izvršavanja
- Statička pohrana za lokalne varijable
- Pristup memoriji i resursima procesa, koje dijele sve niti unutar procesa

Slika 2 prikazuje razliku između niti i procesa sa stajališta upravljanja procesima. U procesu s jednom niti, reprezentacija procesa uključuje kontrolni blok procesa, korisnički adresni prostor kao i stogove za korisnički te jezgreni način rada. U višenitnom okruženju, još uvijek postoji samo jedan kontrolni blok procesa i korisnički adresni prostor, no stogovi su odvojeni za svaku nit pojedinačno. Stoga, sve niti procesa dijele njegovo stanje i resurse te istovremeno vide rezultate djelovanja druge niti. Ako jedna nit otvori datoteku za čitanje, sve ostale niti imaju to ovlaštenje. Ključni benefiti korištenja niti proizlaze iz implikacija performansi:

- Potrebno je puno manje vremena za kreiranje niti unutar procesa nego za kreiranje cijelog novog procesa (oko 10 puta za UNIX)
- Potrebno je manje vremena za prekid niti nego za prekid procesa
- Potrebno je manje vremena za raspoređivanje niti unutar procesa, nego za raspoređivanje procesa
- Niti poboljšavaju efikasnost u komunikaciji između programa; tj. niti unutar istog procesa mogu komunicirati bez djelovanja jezgre jer dijele istu memoriju i podatke.



Slika 2. Modeli procesa s jednom i više niti

Prednosti višenitnog načina pisanja aplikacija su očite i takav način se preporuča u smislu postizanja bolje efikasnosti. Primjer aplikacije koja bi se mogla napisati višenitno je datotečni poslužitelj. Sa svakim novim zahtjevom za nekom datotekom, mogla bi se kreirati nova nit. S obzirom da poslužitelj rukuje s mnogo zahtjeva, velik broj niti će biti kreirano i uništeno u kratkom periodu. Ako se poslužitelj pokreće na višeprocesorskom računalu, moguće je izvođenje više niti unutar tog jednog procesa. U konačnici, s obzirom da niti dijele pristup

datotekama i postoji koordinacija između njihovih radnji, brže je koristiti niti nego višestruke procese. Najčešći primjeri korištenja niti kod jedno-korisničkih višeprocesorskih sustava su:

- Kombinacija blokirajućih i pozadinskih poslova – npr. Excel program u kojem jedna nit prikazuje meni i obrađuje korisnički unos, druga nit vrši kalkulacije i ažurira tablicu
- Asinkrona obrada – primjerice, program za zaštitu od nestanka struje; jedna nit služi za obradu teksta koji ispisuje sadržaj iz RAM-a na disk svake minute, a druga nit je zadužena za periodično kreiranje sigurnosne kopije podataka.
- Brzina čitanja – više niti istovremeno primaju podatke s I/O uređaja
- Modularna programska struktura – programi koji uključuju raznolike aktivnosti i različite izvore te destinacije mogu imati korist od višenitne implementacije.

1.1.1. Stanja niti

U operacijskom sustavu koji podržava niti, raspoređivanje i izvršavanje se čini na razini niti. Također postoje i radnje koje su zajedničke svim nitima, npr. razmjena ili prekidanje procesa. Kao i procesi, niti također mogu biti u nekoliko stanja izvršavanja:

- Pokrenuta (engl. *Spawn*) – prilikom pokretanja procesa ili prilikom eksplicitnog kreiranja nove niti
- Blokirana (engl. *Block*) – kada nit čeka na događaj
- Deblokirana (engl. *Unblock*) – kada se dogodio događaj na kojeg je čekala
- Završena (engl. *Finish*) – kada nit završi, sadržaj njenih registara i stogova se oslobađa

1.2. Vrste niti

Kada je riječ o implementaciji niti, postoje dvije široke implementacijske kategorije: **niti na korisničkoj razini** (engl. *user-level threads*) i **niti na razini jezgre** (engl. *kernel-level threads*). Kod niti na korisničkoj razini, sav posao upravljanja nitima vrši aplikacija, a jezgra nije svjesna postojanja tih niti. Svaka aplikacija se može programirati na višenitni način, koristeći biblioteke koje omogućavaju rad s nitima. Kod niti na razni jezgre, sav posao upravljanja nitima obavlja jezgra operacijskog sustava. Ne postoji kod za upravljanje nitima unutar aplikacija, nego se koristi programsko sučelje (engl. *application programming interface (API)*) prema uslugama jezgre. Oba načina nude brojne prednosti, ali i nedostatke.

Uobičajeno, aplikacija započinje s jednom niti u kojoj se u potpunosti izvršava. U nekom trenutku, aplikacija može stvoriti novu nit unutar istog procesa. Biblioteka za upravljanje nitima kreira potrebne strukture podataka za novu nit te prosljeđuje kontrolu jednoj od spremnih niti unutar procesa, po određenom algoritmu za raspoređivanje. Cijela aktivnost se odvija na korisničkoj razini unutar jednog procesa, a jezgra nije svjesna njenog odvijanja. Prednosti niti na korisničkoj razini su:

- Razmjena niti (engl. *thread switching*) ne zahtijeva intervenciju jezgre, jer su sve podatkovne strukture potrebne za upravljanje niti sadržane unutar korisničkog adresnog prostora procesa.
- Raspoređivanje je specifično za aplikaciju - sam programer aplikacije odlučuje o raspoređivanju niti te mu nije potreban raspoređivač operacijskog sustava.
- Korisničke niti su portabilne za svaki operacijski sustav

Međutim korisničke niti nose i određene nedostatke:

- Mogućnost blokiranja cijelog procesa - ako jedna nit napravi sustavski poziv prema jezgri, blokira se cijeli proces (s obzirom da se jezgra ne razlikuje korisničke niti nego gleda njima pripadni proces)

- Višenitne aplikacije koje upotrjebljuju korisničke niti ne mogu u potpunosti iskoristiti prednost višestruke obrade (engl. *multiprocessing*). Jezgra u jednom trenutku procesoru pridružuje samo jedan proces, tj. jednu korisničku nit.

Ukratko, korisničke niti su konkurentne unutar aplikacije, tj. procesa, ali se na razini operacijskog sustava opet promatraju kao procesi koji se ne mogu istovremeno izvršavati.

U višenitnoj aplikaciji, zasnovanoj na nitima na razini jezgre, sav posao upravljanja nitima preuzima jezgra operacijskog sustava. Raspoređivanje se odrađuje na razini niti i tako se savladavaju oba nedostatka niti na korisničkoj razini. Prvo, jezgra može simultano raspoređivati više niti iz jednog procesa na više procesora. Drugo, ako je jedna nit blokirana, jezgra može izvršavati ostale niti unutar procesa. S druge strane, temeljni nedostatak niti na razini jezgre je što prijenos kontrole među nitima zahtijeva prebacivanje načina rada iz korisničkog u jezgreni. Naravno, takav prijenos troši mnogo vremena procesora, pa su niti na razini jezgre sporije od korisničkih niti.

2. POSIX BIBLIOTEKA ZA UPRAVLJANJE NITIMA

POSIX je akronim od *Portable Operating System Interface*, što je IEEE standard dizajniran za olakšavanje portabilnosti aplikacija. Nastao je kao pokušaj konzorcija dobavljača u stvaranju jedinstvene verzije operacijskog sustava UNIX. Što više distribucija tog operacijskog sustava slijedi POSIX standard, lakše je prenijeti aplikacije između različitih hardverskih platformi. Postoji nekoliko dijelova POSIX standarda, pri čemu POSIX.1 definira sučelja programskog jezika C za rad s datotekama, procesima i I/O uređajima, dok POSIX.2 definira naredbe ljuske. Jedno od aplikacijskih sučelja (engl. *application programming interface (API)*), definiranih u POSIX.1c je *POSIX Threads API*, koji se uobičajeno naziva i **pthreads**. Pthreads je API koji olakšava kontrolu višestrukih tokova izvršavanja unutar aplikacije. Svaki tok izvršavanja se predstavlja nitima, pa stoga ovaj API olakšava kreiranje i upravljanje nitima. API je neovisan o programskom jeziku, no inicijalno je napisan za C/C++ jezike. Postoji niz različitih implementacija POSIX Threads API-ja, za različite operacijske sustave.

Pthreads API definira skup tipova, funkcija i konstanti u programskom jeziku C. Implementiran je kao biblioteka *pthread.h*. Postoji više od 100 procedura biblioteke, pri čemu sve započinju s prefiksom *pthread_*, a mogu se kategorizirati u četiri grupe:

- Upravljanje nitima - kreiranje, spajanje niti i sl.
- Muteksi
- Uvjetne varijable
- Sinkronizacija niti pomoću barijera i zaključavanja na čitanje/pisanje

2.1. Kreiranje i uklanjanje niti

Unutar Pthreads API-ja, nit je reprezentirana tipom podatka **pthread_t**. Za kreiranje niti, koristi se funkcija **pthread_create**:

```
int pthread_create(pthread_t * thread,
                   const pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void *arg);
```

Argumenti funkcije su:

- *thread* - id novostvorene niti
- *attr* - zastavice iz strukture *_attr_t*
- *start_routine* - pokazivač na funkciju koju nit treba izvesti
- **arg* - pokazivač na argumente funkcije koju nit izvodi

Za uklanjanje niti, koristi se funkcija **pthread_exit**, koja prima povratnu vrijednost niti.

```
void pthread_exit(void *retval);
```

2.2. Čekanje na završetak niti

Funkcija **pthread_exit** završava nit i ne vraća povratnu vrijednost. Na taj način, druge niti ne mogu znati je li ta nit stvarno "uništena". Ako su niti međusobno zavisne, takva informacija je vjerojatno vrlo bitna ostalim nitima. Stoga, funkcija **pthread_join** ispituje id i povratnu vrijednost niti iz neke druge niti.

```
int pthread_join(pthread_t thread, void **retval);
```

Također, proces u kojem se niti stvaraju može čekati na njihov završetak (npr. *main* funkcija). U suprotnom se može dogoditi da proces završi dok se njegove niti još izvršavaju. U tom slučaju, niti procesa također bivaju uklonjene. Primjer 1 prikazuje jednostavan program u kojem su kreirane dvije niti koje izvršavaju istu funkciju s različitim argumentima. Glavni proces, tj. *main* funkcija, čeka na završetak obje niti, pa zatim ispisuje njihove povratne vrijednosti. Bitno je napomenuti da se program kompajlira uz opciju ***-lpthread***. Primjerice: *gcc thread.c -o thread -lpthread*.

Primjer 1: Jednostavan primjer višenitnog programa

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message_function ( void *ptr );
main () {
  pthread_t thread1, thread2;
  char *message1 = "Thread 1";
  char *message2 = "Thread 2";
  int ret1, ret2;
  // Kreiranje dviju neovisnih niti koje izvršavaju istu funkciju s različitim argumentom
  ret1 = pthread_create (&thread1, NULL, print_message_function, (void*) message1);
  ret2 = pthread_create (&thread2, NULL, print_message_function, (void*) message2);

  // Čekanje na završetak obje niti
  pthread_join (thread1, NULL);
  pthread_join (thread2, NULL);
  printf("Thread 1 vraća :%d\n", ret1);
  printf("Thread 2 vraća: %d\n", ret2);
  exit(0);
}
void *print_message_function (void *ptr) {
  char *message;
  message = (char *) ptr;
  printf("%s \n", message);
}
```

3. UPRAVLJANJE NITIMA UNUTAR OPERACIJSKOG SUSTAVA WINDOWS

Nit je osnovna jedinica izvršavanja u operacijskom sustavu Windows, a proces može sadržavati više neovisnih niti koje dijele adresni prostor i druge resurse procesa. Višenitno programiranje zahtijeva organizaciju i koordinaciju izvođenja niti kako bi se pojednostavili programi i iskoristile prednosti paralelizma.

3.1. Kreiranje niti

Kao i kod procesa, niti su predstavljene objektima jezgre te postoje sustavski pozivi za njihovo upravljanje. Sustavski poziv *CreateThread* kreira izvršnu niti unutar adresnog prostora procesa koji ga izvodi. Funkcija *CreateThread* omogućava specificiranje točne adrese unutar adresnog prostora procesa za smještanje niti te veličinu stoga.

```
HANDLE CreateThread (LPSECURITY_ATTRIBUTES lpsa, DWORD dwStackSize,
LPTHREAD_START_ROUTINE lpStartAddr, LPVOID lpThreadParm, DWORD dwCreationFlags,
LPDWORD lpThreadId)
```

Parametri funkcije *CreateThread* su:

- *lpsa* - struktura podataka za sigurnosne atribute
- *dwStackSize* - veličina stoga niti u bajtovima
- *lpStartAddrs* - pokazivač na funkciju koju nit izvodi
- *lpThreadParm* - pokazivač na argumente funkcije koju nit izvodi
- *dwCreationFlags* - zastavice koje ukazuju na stanje novokreirane niti
- *lpThreadId* - 32-bitni pozitivni cijeli broj koji predstavlja ID novokreirane niti

3.2. Uklanjanje niti

Sve niti unutar procesa mogu završiti pomoću funkcije *ExitThread*. U tom slučaju, stog niti se dealocira iz memorije i oslobađaju se svi *handle* objekti koje je ta nit kreirala.

```
VOID ExitThread (DWORD dwExitCode)
```

Kada se ukloni zadnja niti unutar nekog procesa, tada i proces završava s radom. Jedna nit može zatvoriti drugu nit pozivom funkcije *TerminateThread*.

3.3. Korištenje niti u programskom jeziku C

Primjer 2 prikazuje programski kod za kreiranje niti koristeći operacije za upravljanje nitima unutar Win32 API-ja. Prije pozivanja funkcije *CreateThread*, potrebno je definirati funkciju koju nit treba izvoditi. U ovom primjeru nit izvodi funkciju koja kvadrira elemente niza cijelih brojeva.

Primjer 2: Programski kod za kreiranje niti

```
#include<Windows.h>
#include<stdio.h>
using namespace std;

int polje[] = {1, 2, 3, 4, 5};
DWORD WINAPI ThreadFunction (LPVOID lpParam) {
  int i;
  for (i=0; i<5; i++){
    polje[i] = polje[i] * polje[i];
    printf("%d ", polje[i]);
  }
  return 0;
}
int main()
{
  HANDLE hThread;
  DWORD ThreadID;

  hThread = CreateThread (NULL, 0, ThreadFunction, (void*) &polje, 0, &ThreadID);

  if (hThread == NULL){
    printf("Neuspjelo kreiranje niti, Error :%d\n", GetLastError());
  } else {
    printf("Uspjesno kreiranje niti\n");
  }
  CloseHandle(hThread);
  system("PAUSE");
  return 0;
}
```

4. KONTROLA ISTOVREMENOSTI MEĐUSOBNIM ISKLJUČIVANJEM

Dizajniranje ispravnih rutina za kontrolu konkurentnih aktivnosti jedan je od najtežih aspekata sistemskog programiranja. **Istovremenost** (engl. *concurrency*) je zajednička svim modernim konceptima koji pokušavaju podijeliti posao na nekoliko jedinica izvršavanja, tj. *multiprogramming*, *multiprocessing* i *distributed processing* konceptima. Ona obuhvaća mnoštvo koncepata operacijskih sustava, kao što su: međuprocesna komunikacija, dijeljenje resursa, sinkronizacija više procesa i alokacija procesora. Istovremenost se ostvaruje u tri aspekta:

- Više istovremenih aplikacija - *multiprogramming* koncept je uveden kako bi omogućio dijeljenje vremena procesora između više aplikacija
- Strukturirane aplikacije - aplikacije mogu biti efektivno programirane kao skup konkurentnih procesa (niti)
- Struktura operacijskog sustava - operacijski sustavi su također implementirani kao nizovi procesa i niti

4.1. Principi istovremenosti

4.1.1. Uvjeti utrke

Uvjeti utrke (engl. *race conditions)* se javljaju kada više procesa (ili niti) želi čitati ili pisati iste strukture podataka, a finalni rezultat ovisi o redoslijedu njihovih izvršavanja. Primjer 3 prikazuje pojednostavljen primjer programa u kojem više niti može izvršavati sljedeći dio koda nad zajedničkom strukturom podataka *counter*. Ako niti paralelno koriste vrijednost varijable *counter*, od provjere stanja (*if*) pa sve do promjene vrijednosti (*counter = counter -1*) može se svašta dogoditi s vrijednosti varijable. Primjerice, ako je početna vrijednost varijable *counter* bila 1, obje niti mogu paralelno ući u uvjet te izvesti određeni zadatak te smanjiti vrijednost brojača. Međutim, jedna od njih će to napraviti brže, pa će druga svoj zadatak obavljati nad izmijenjenom vrijednosti varijable, a neće biti toga ni svjesna.

Primjer 3: Jednostavan primjer problema s paralelnim radom niti

```
…
if (counter > 0) {
   // do task
   counter = counter – 1;
}
```

U slučaju utrke procesa za istim resursom, uvodi se posebna terminologija. Tako se resurs, za kojim se natječu procesi, naziva **kritični resurs** (engl. *critical resource*). Dio programskog koda koji želi pristupiti kritičnom resursu se naziva **kritični odsječak** (engl. *critical section*). U nekom trenutku, samo jedan proces smije biti u svom kritičnom odsječku. Za primjer 3 kritični odsječak je cijeli uvjet, a kritični resurs je varijabla *counter*.

4.1.2. Problemi kontrole istovremenosti

Moguće je klasificirati načine na koji procesi međusobno djeluju jedan na drugoga, ovisno o stupnju svjesnosti nekog procesa o postojanju drugih procesa. Tablica 1 prikazuje moguće

stupnje svjesnosti te posljedice interakcije takvih procesa u vidu problema s kontrolom istovremenosti. Poznati problemi kontrole istovremenosti su:

- **Međusobno isključivanje** (engl. *mutual exclusion*) - osiguravanje da je samo jedan proces unutar svog kritičnog odsječka u slučaju utrke.
- **Zastoj** (engl. *deadlock*) - izvršavanje međusobnog isključivanja može uzrokovati zastoj. Primjerice, ako 2 procesa istovremeno čekaju na 2 resursa, a operacijski sustav svakom da jedan od resursa. Tada su oba procesa u zastoju jer su im potrebna oba resursa za daljnji rad. Nijedan proces neće otpustiti resurs kojim raspolaže pa time neće dobiti ni pravo na drugi resurs.
- **Gladovanje** (engl. *starvation*) - također problem kontrole prouzrokovan izvršavanjem međusobnog isključivanja. Pretpostavimo da tri procesa (P1, P2, P3) zahtijevaju periodički pristup nekom resursu i da se nakon izvršavanja procesa P1, resurs dodijeli procesu P3. Ako se odmah nakon izvršavanja procesa P3 ponovno pojavi P1 i opet mu se dodijeli vlasništvo nad resursom, tada je P2 u stanju gladovanja. Sve dok se potpuno ne izvrše procesi P1 i P3, P2 neće dobiti vlasništvo nad resursom.

Tablica 1. Vrste međuprocesne interakcije

Stupanj svjesnosti	Odnos	Utjecaj jednog procesa na drugi	Potencijalni problemi kontrole
Procesi su nesvjesni o postojanju drugih procesa	Natjecanje	Rezultat jednog procesa je neovisan o akcijama drugih procesa	• Međusobno isključivanje • Zastoj • Gladovanje
Procesi su indirektno svjesni o postojanju drugih procesa (npr. preko zajedničkog objekta)	Kooperacija putem dijeljenja	Rezultat jednog procesa ovisi o informaciji dobivenoj od drugih procesa	• Međusobno isključivanje • Zastoj • Gladovanje • Koherentnost podataka
Procesi su direktno svjesni o postojanju drugih procesa (zajednički komunikacijski mehanizmi)	Kooperacija putem komunikacije	Rezultat jednog procesa ovisi o informaciji dobivenoj od drugih proces	• Zastoj • Gladovanje

4.1.3. Međusobno isključivanje

Osnovni preduvjet ostvarivanja konkurentnosti 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. Uspješno rješenje za problem međusobnog isključivanja ima 2 svojstva:

- Uspješno osigurava međusobno isključivanje - samo jedan proces u kritičnom odsječku
- Ne dolazi do zastoja - jedan proces s vremenom mora ući u kritični odsječak, tj. ne smije se dogoditi da svi procesi postaju blokirani.

4.2. Algoritmi međusobnog isključivanja

Postoji nekoliko algoritama i tehnika kojima se implementira međusobno isključivanje. Četiri su zahtjeva koje ti algoritmi moraju zadovoljiti:

1. U kritičnom odsječku u svakom trenutku smije biti najviše jedna nit.
2. Mehanizam međusobnog isključivanja mora djelovati i u uvjetima kada su brzine izvođenja niti proizvoljne.
3. Kada neka od niti zastane u svom nekritičnom dijelu ona ne smije spriječiti ulazak druge niti u svoj kritični odsječak.
4. Izbor jedne o niti koja smije ući u kritični odsječak treba obaviti u konačnom vremenu.

Međusobno isključivanje je moguće postići i hardverskim i softverskim putem. Najjednostavnije hardversko rješenje za nametanje međusobnog isključivanja je onemogućenje prekida (engl. *interrupts*) kada je proces u kritičnom odsječku. Iako je rješenje efektivno, može prouzrokovati predugo čekanje drugih procesa. Štoviše, u slučaju ako se proces u izvršavanju zaustavi (npr. čeka na I/O uređaj), ne može se prepustiti procesor nekom drugom procesu. Elegantnije rješenje za postizanje međusobnog isključivanja je 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.

4.2.1. Dekkerov algoritam

Nizozemski matematičar Theodorus Dekker prvi je osmislio elegantno softversko rješenje međusobnog isključivanja. Njegov algoritam je protumačio i promovirao Edsger Dijkstra 1965. godine. Algoritam se temelji na zastavicama (engl. *flags*), varijablama koje mogu poprimiti samo vrijednosti 0 ili 1. Pri tome, kritični odsječak je slobodan kada je vrijednost zastavice jednaka 0. U suprotnome, jedna nit je unutar kritičnog odsječka, a nit koja želi ući u njega mora čekati promjenu vrijednosti te varijable. Dekkerov algoritam radi samo za dvije konkurentne niti, pri čemu svaka nit ima vlastitu zastavicu. U slučaju obostrana zahtjeva za ulazak, nit koja nije na redu će spustiti svoju zastavicu i pričekati drugu nit da završi sa svojim kritičnim odsječkom. Ukoliko obje niti imaju podignutu zastavicu, tj. žele ući u kritični odsječak, izabire se nit većeg prioriteta.

Ovaj algoritam se zasniva na tehnici međusobnog isključivanja čekanjem, tako što se jedna nit nalazi u petlji koja s ciklusom takta procesora provjerava je li druga nit izašla iz kritičnog odsječka. Nit *i* želi ući u kritički odsječak i odmah diže zastavicu. Međutim, ukoliko je trenutno red na drugu nit *j* (što je iskazano varijablom *turn*), prva nit spušta svoju zastavicu i čeka dok druga nit ne napusti kritični odsječak. Iako ovaj algoritam uspješno nameće međusobno isključivanje, konstantno provjeravanje vrijednosti zastavica uvelike troši procesorsko vrijeme. Osim toga, Dekkerov algoritam nije proširiv za više niti.

Primjer 4: Procedure Dekkerova algoritma

```
…
bool flags[2];
bool turn;
…
void enter_critical_section (i) {   //i je indeks niti, može biti prva (0) ili druga (1) nit
    j = 1 – i;
    flags[i] = 1;
    while (flags[j] != 0) {
        if (turn == j) {
           flags[i] = 0;
           while (turn == j);
           flags[i] = 1;
       }
   }
  // neće izaći iz petlje dok se flags[j] ne spusti
}

void leave_critical_section(i) {
    flags[i] = 0;
    turn = 1 – i;
}
```

4.2.2. Petersonov algoritam

Godine 1981. Gary Peterson otkrio je mnogo jednostavniji način ostvarivanja međusobnog isključivanja čekanjem. Prema Petersonu, u slučaju obostrana zahtjeva za ulazak u kritički odsječak, nit koja je došla kasnije daje prednost onoj drugoj i čeka da ona završi sa svojim izvođenjem. Procedure Petersonova algoritma su prikazane u Primjeru 5.

Primjer 5: Procedure Petersonova algoritma

```
…
bool flags[2];
bool turn;
…
void enter_critical_section (i) {   //i je indeks niti, može biti prva (0) ili druga (1) nit
    j = 1 – i;
    flags[i] = 1;
    turn = j;
    while (flags[j] != 0 && turn == j);
}

void leave_critical_section(i) {
    flags[i] = 0;
}
```

U odnosu na Dekkerov algoritam, Petersonov algoritam je kraći i brži. Osim toga, ne ovisi o početnoj vrijednosti varijable *turn*. Međutim, i ovaj algoritam također radi samo za sustave sa samo dvije niti.

4.2.3. Lamportov (pekarski) algoritam

Još jedan od poznatijih softverskih rješenja međusobnog isključivanja je Lamportov algoritam koji je ostvariv za dvije ili više niti. Pri dizajniranju svog algoritma, Leslie Lamport se oslonio na analogiju pekare u kojoj na ulazu postoji stroj za prebrojavanje kupaca koji svakom kupcu dodjeljuje jedinstveni redni broj. Istovremeno, stroj na zajedničkom ekranu prikazuje broj kupca koji se trenutno poslužuje. Pri tome, svi drugi kupci čekaju u redu sve dok pekar ne završi s posluživanjem trenutne mušterije. Po završetku kupovine, na zajedničkom ekranu se prikazuje broj idućeg kupca kojeg pekar mora poslužiti. Ukoliko se već posluženi kupac sjeti da nije završio s kupovinom, on mora ponovno uzeti svoj broj iz automata za prebrojavanje. Prema ovoj analogiji, kupci predstavljaju niti u računalnom sustavu koje se identificiraju brojevima koje im dodijeli sustav. Pravo na ulazak u kritični odsječak ima nit s najmanjim dodijeljenim brojem. Ako nekim slučajem dvije niti dobiju isti broj, onda se gleda i indeks niti. Sam postupak dodjeljivanja brojeva je također na neki način kritični odsječak (zamislite da više kupaca „istovremeno“ zatraži svoj broj od stroja za prebrojavanje), pa se i on ograđuje (pomoću polja *enters*). Primjer 6 prikazuje procedure Lamportova algoritma.

Primjer 6: Procedure Lamportova algoritma

```
…
int numbers[n];  // brojevi koji identificiraju niti
int enter[n];   // brojevi koji štite redoslijed dodjeljivanja identifikatora niti
…
void enter_critical_section (i) {
    enter[i] = 1;
    numbers[i] = max(numbers) + 1;
    enter[i] = 0;

   for (int j=1; j<n; j++) {   // provjera i čekanje na nit s manjim brojem
      while (enter[j] == 1);  // čekanje da nit j dobije broj, ako je u postupku dobivanja
      while (enter[j] != 0 && (enter[j] < enter[i] || (enter[j] == enter[i] && j < i)));
     // čekaj ako nit j ima prednost
   }
}

void leave_critical_section(i) {
    numbers[i] = 0;
}
```

5. ZADACI ZA SAMOSTALNI RAD

Pristupiti poslužitelju linux.etfos.hr pomoću Putty SSH klijenta. Sljedeće zadatke možete riješiti 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.** Druga opcija rješavanja je korištenje MS Visual Studio okruženja ili online kompajlera. Za svaki zadatak kopirajte programski kod, naredbe ljuske i dobiveni izlaz u Zadaću. Ako je potrebno, napišite dodatno pojašnjenje Vašeg rješenja.

1. Koristeći POSIX Threads biblioteku kreirajte program koji kreira i pokreće 3 neovisne niti: jednu nit za ispis najvećeg elementa polja, drugu za ispis najmanjeg elementa polja i treću za ispis cijelog polja. Polje inicijalizirajte kao globalnu varijablu.
2. Koristeći biblioteke Win32 API-ja napravite program koji kreira i pokreće 3 neovisne niti: jednu nit za ispis najvećeg elementa polja, drugu za ispis najmanjeg elementa polja i treću za ispis cijelog polja. Polje inicijalizirajte kao globalnu varijablu. Pokrenite program u MS Visual Studio okruženju ili pomoću online kompajlera te komentirajte ima li razlike u ispisu u odnosu na Zadatak 1.
3. Koristeći biblioteku za upravljanje nitima po volji (POSIX Threads ili Win32 API) napravite program koji kreira i pokreće 2 niti: prva nit uvećava i ispisuje vrijednost neke globalne varijable za 5, dok druga nit smanjuje i ispisuje vrijednost te iste varijable za 3.
4. Program iz zadataka 3 nadopunite tako da implementirate međusobno isključivanje dviju niti pomoću Dekkerova algoritma.
5. Program iz zadataka 3 nadopunite tako da implementirate međusobno isključivanje dviju niti pomoću Petersonova algoritma.

5.1. 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. Nadopunjavanje proizvoljnog višenitnog programa implementiranjem međusobnog isključivanja pomoću Dekkerova algoritma.
2. Nadopunjavanje proizvoljnog višenitnog programa implementiranjem međusobnog isključivanja pomoću Petersonova algoritma.
3. Nadopunjavanje proizvoljnog višenitnog programa implementiranjem međusobnog isključivanja pomoću Lampertova algoritma.

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] Hart, J.M., 2010. Windows system programming. Pearson Education.

[7] 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)

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

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