

# OPERATIVNI SISTEMI

Slajdovi su kreirani na osnovu knjige “Operativni sistemi, principi unutrašnje organizacije i dizajna, 7. izdanje“, William Stallings, CET, Beograd, 2013.

Konkurentnost – međusobna isključivost i sinhronizacija

# Međusobno nezavisne niti



izvor: [www.youtube.com](http://www.youtube.com)

# Međusobno zavisne niti – deljenje resursa



izvor: [www.youtube.com](http://www.youtube.com)

# Međusobno zavisne niti – sinhronizacija



izvor: [www.youtube.com](http://www.youtube.com)

# Upravljanje procesima

- OS može pri upravljanju procesima da omogući
  - ▣ Multiprogramiranje
    - više procesa unutar jednog procesorskog sistema
  - ▣ Multiprocesiranje
    - više procesa unutar multiprocesora
  - ▣ Distribuiranu obradu
    - više procesa na više distribuiranih računara
- Konkurentnost
  - ▣ važan aspekt pri svakom upravljanju višestrukim procesima
  - ▣ pitanja međusobne interakcije i zavisnosti procesa

# Gde se javlja konkurentnost?

- Svugde gde se vrši preplitanje različitih procesa
  - ▣ kompletan scenario i vremenski trenutak preplitanja je u pravilu nepredvidiv
- Višestruke aplikacije
  - ▣ kod multiprogramiranih sistema različiti procesi pristupaju istim resursima
- Strukturirane korisničke aplikacije
  - ▣ jedna aplikacija može da sadrži više konkurentnih procesa/niti
- Operativni sistem
  - ▣ funkcije OS su implementirane kao više procesa/niti

# Problemi koje konkurentnost donosi

- Deljenje globalnih resursa
  - ▣ Štetno preplitanje
    - Dva procesa vrše upis/čitanje nad istom globalnom promenljivom
    - Neodređen redosled izvršavanja i rezultata operacija
- Komplikovanja dodela resursa
  - ▣ Resurs može biti dodeljen procesu koji ga ne koristi
  - ▣ Procesi mogu biti međusobno blokirani zbog čekanja na resurse
- Detekcija grešaka
  - ▣ Teže je utvrditi zašto se program neočekivano ponaša
  - ▣ Rezultati nisu deterministički i teže je reprodukovati neočekivano ponašanje

# Kako se problemi rešavaju

- Međusobna isključivost
  - ▣ mogućnost procesa da obavi akciju bez negativnog uticaja drugih procesa
- Sinhronizacija
  - ▣ usklađivanje ponašanja procesa sa aktivnošću drugih procesa



# Primer štetnog preplitanja

```
1.  int x;  
2.  void echo() {  
3.      cin >> x;  
4.      int y = x;  
5.      cout << y;  
6.  }
```

- Dva procesa pozivaju metodu
- x je deljeni resurs
- Neispravan rad ako se desi preplitanje nakon naredbe u liniji 3
- Problem se može rešiti ako se odredi da samo jedan proces u datom trenutku može izvršavati kod metode

# *Data race vs determinizam*

## □ *Data race*

- ▣ Rezultat programa zavisi od scenarija preplitanja i redosleda izvršavanja instrukcija između procesa
- ▣ Ponovna izvršavanja istog koda ne daju isti rezultat

## □ Determinističko ponašanje

- ▣ Rad procesa mora biti nezavisan od brzine izvršavanja relativne u odnosu na brzinu ostalih tekućih procesa

# Data race - primer

---

- Primeri/Konkurentnost/Sum

# Uzajamno delovanje procesa

- Procesi nisu svesni drugih procesa
  - ▣ Nezavisni procesi koji nisu predviđeni da rade zajedno
  - ▣ Ipak, pristupaju istim resursima
  - ▣ Nadmetanje za resurse
- Procesi su svesni drugih procesa
  - ▣ Procesi projektovani da zajednički obave posao
  - ▣ Eksplicitna sinhronizacija aktivnosti

# Nadmetanje procesa za resurse

- Uzajamna isključivost procesa
  - ▣ Više procesa treba da koristi isti resurs
  - ▣ Za ispravan rad potrebno je da u jednom trenutku samo jedan proces pristupa resursu
  - ▣ Ovakav resurs nazivamo kritični resurs



# Nadmetanje procesa za resurse

## □ Kritična sekcija

- Deo programa u kojem se pristupa kritičnom resursu
- Za ispravan rad, kod u kritičnoj sekciji procesi moraju da izvršavaju sekvencijalno (jedan po jedan)
- U jednom trenutku samo jedan proces sme da bude u kritičnoj sekciji



# Ulazak u kritičnu sekciju

- Ako je jedan proces ušao u kritičnu sekciju
  - ▣ šta se dešava sa drugim procesima koji pokušavaju da uđu?

# Prva varijanta – *busy waiting*

- Proces koji ne može da uđe u kritičnu sekciju ostaje aktivan neprekidno proveravajući da li može da uđe
- Proces nastavlja da troši procesorsko vreme i kada nema uslova da radi





# Prva varijanta – *busy waiting*

```
int x;
bool is_lock_free = true;
void echo() {
    while (!is_lock_free) {
    }
    is_lock_free = false;
    cin >> x;
    int y = x;
    is_lock_free = true;
    cout << y;
}
```

# Druga varijanta – blokiranje procesa

- Proces koji ne može da uđe u kritičnu sekciju odlazi u stanje blokiran
- Sistem ga obaveštava kada može da uđe u kritičnu sekciju



# Druga varijanta – blokiranje procesa

```
int x;  
mutex m;  
void echo() {  
    m.lock();  
    cin >> x;  
    int y = x;  
    m.unlock();  
    cout << y;  
}
```

Propusnica za kritičnu sekciju

- Uzimanje propusnice
- Proces proverava da li je propusnica slobodna
- Ako jeste nastavlja
- Ako nije odlazi u stanje blokiran

- Vraćanje propusnice
- Jedan od procesa koji čekaju na propusnicu prelazi u stanje spreman

# Propusnica

## □ Lock

- Svaki proces pre ulaska u kritičnu sekciju zatraži zaključavanje nekog **deljenog** objekta
- Proces koji prvi zatraži zaključavanje, uspeva da zaključa objekat i ulazi u kritičnu sekciju
- Ovo zaključavanje je implementirano uz oslonac na hardversku podršku za uzajamnu isključivost
- Svaki naredni proces ne uspeva da zaključa objekat i prelazi u stanje blokiran
- Proces pri izlasku iz kritične sekcije otključava objekat
- Jedan od procesa koji čekaju, uspeva da zaključa objekat i ulazi u kritičnu sekciju
- Da bi ovo radilo, svi procesi moraju da zaključavaju **isti** objekat (ne sme svaki proces raditi sa svojom lokalnom kopijom)

# Kako se implementira zauzimanje propusnice?

- Pri uzimanju propusnice potrebno je
  - ▣ Proveriti da li je propusnica slobodna
    - Ako jeste, zauzeti propusnicu
    - Ako nije, postaviti proces u stanje blokiran

```
1.  if (is_lock_free)
2.      is_lock_free = false;
```

- Šta ako se nakon linije 1 desi preplitanje?
- Dva procesa mogu da utvrde da je propusnica slobodna i da oba uđu u kritičnu sekciju
- Zato se zauzimanje propusnice ne implementira na ovaj način, već se oslanja na hardversku podršku za uzajamnu isključivost

# Hardverska podrška za uzajamnu isključivost

- Onemogućavanje prekida
  - ▣ Proces se izvršava do prekida
  - ▣ Ako onemogućimo prekide pri zauzimanju propusnice, ne može doći do preplitanja
  - ▣ Neefikasno
    - procesor ograničen da prepliće programe
  - ▣ Ne radi na multiprocesorskoj arhitekturi
    - procesi se izvršavaju na različitim procesorima
    - onemogućavanje prekida na jednom procesoru ne sprečava drugi proces da pristupi resursu

# Hardverska podrška za uzajamnu isključivost

- Specijalne mašinske instrukcije
  - ▣ Hardver obezbeđuje instrukcije koje obavljaju više operacija atomski (nedeljivo)
  - ▣ Ove instrukcije se mogu iskoristiti za zauzimanje propusnice
  - ▣ Proces u jednom nedeljivom koraku proverava
    - da li može da zauzme propusnicu  $i$
    - ako može, postavlja indikator da je zauzeo

# Hardverska podrška za uzajamnu isključivost

- Instrukcija Compare&Swap
  - ▣ Opis instrukcije u jeziku visokog nivoa

```
int compare_and_swap(int* word, int testval, int newval)
{
    int oldval = *word;
    if (oldval == testval)
        *word = newval;
    return oldval;
}
```

- Instrukcija nedeljivo izvršava skup operacija
- Proverava se trenutna vrednost
- Postavlja se nova vrednost ako trenutna vrednost ispunjava uslov



# Hardverska podrška za uzajamnu isključivost

- Upotreba instrukcije Compare&Swap za obezbeđenje međusobne isključivosti
- Zaključava se objekat pri ulasku u kritičnu sekciju
- Ilustracija ponašanja u jeziku višeg nivoa

```
if (compare_and_swap(locked_flag, 0, 1) == 0) {  
    // proces nastavlja rad  
} else {  
    // proces ide na cekanje  
}
```

# Hardverska podrška za uzajamnu isključivost

## □ Instrukcija Exchange

### ▣ Opis instrukcije u jeziku višeg nivoa

```
void exchange(int* register, int* memory) {  
    int temp = *memory;  
    *memory = *register;  
    *register = temp;  
}
```

# Hardverska podrška za uzajamnu isključivost

- Upotreba instrukcije Exchange za obezbeđenje međusobne isključivosti
- Zaključava se objekat pri ulasku u kritičnu sekciju
- Ilustracija ponašanja u jeziku višeg nivoa

```
bool locked = false; //globalna
```

```
bool keyi = true; //lokalna za svaki proces  
exchange(keyi, locked);  
if (keyi == true) {  
    //proces ide na cekanje  
}
```

# Kritična sekcija u C++11 standardu

- Kreiranje kritične sekcije vrši se objektima klase mutex
- Objekat klase mutex je taj deljeni objekat kojeg procesi zaključavaju
- Metode klase mutex
  - ▣ lock()
    - zaključava mutex objekat ako je otključan
    - u suprotnom proces odlazi u čekanje
  - ▣ unlock()
    - otključava mutex objekat
  - ▣ try\_lock()
    - pokušava da zaključa mutex
    - ako ne uspe vraća false, proces ne odlazi u čekanje

# Primer korišćenja klase mutex

---

- Primeri/Konkurentnost/MutexSum

# Kritična sekcija u C++11 standardu

- Pri korišćenju klase mutex postoji opasnost da propusnica ostane zaključana
- Klasa `unique_lock`
  - ▣ Koristi se umesto mutex klase
  - ▣ Upravlja zaključavanjem mutexa
  - ▣ U konstruktoru se kao parametar prosleđuje mutex koji je potrebno zaključati
  - ▣ U destrukturu vrši automatsko otključavanje mutexa
- Korišćenjem objekta klase `unique_lock` propusnica se oslobađa automatski kada objekat prestane da postoji

# Primer – unique\_lock

- 05-Konkurentnost/UniqueLockSum

# Kopiranje mutex objekta

- Objekat klase mutex nije moguće kopirati po vrednosti
- Razlog je da se ne bi desilo da niti zaključavaju različite kopije propusnice
- Realizovano je eksplicitnom naredbom kompajleru da ne generiše konstruktor kopije u klasi mutex

```
class mutex {  
    . . .  
    mutex(const mutex&) = delete;  
    . . .  
}
```



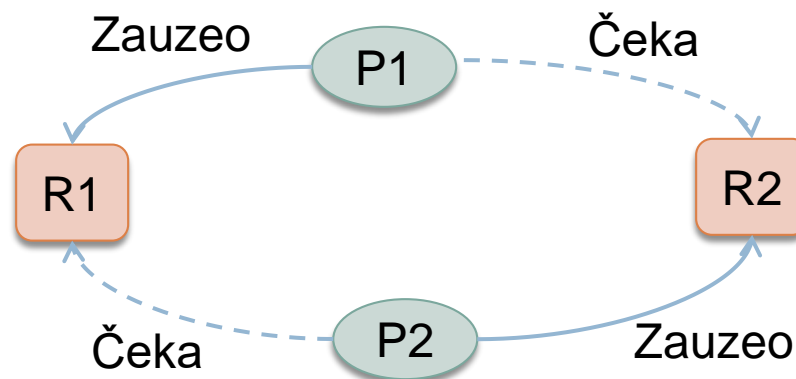
# Nadmetanje procesa za resurse

- Uzajamno blokiranje (*Deadlock*)
  - ▣ Svi procesi međusobno čekaju da drugi procesi oslobode resurse koji im trebaju



# Nadmetanje procesa za resurse

- Uzajamno blokiranje (*Deadlock*)
  - ▣ Procesi P1 i P2 koriste resurse R1 i R2
  - ▣ I P1 i P2 trebaju oba resursa da bi završili posao
  - ▣ Scenario
    - P1 zauzme R1
    - P2 zauzme R2
    - Oba procesa čekaju da onaj drugi proces oslobodi resurs



# Nadmetanje procesa za resurse

## □ *Livelock*

- ▣ Situacija u kojoj dva ili više procesa nisu blokirani, ali menjaju stanja tako da se međusobno onemogućavaju da napreduju
- ▣ Nijedan proces nije blokirani
- ▣ Nijedan proces ne napreduje u vršenju korisnog rada



# Nadmetanje procesa za resurse

## □ Gladovanje

- ▣ Situacija u kojoj proces, iako je spreman, nikad ne dobija procesor od raspoređivača



# Nadmetanje procesa za resurse

## □ Gladovanje

- ▣ Procesi P1, P2 i P3 pristupaju resursu R u kritičnoj sekciji
- ▣ Na početku su sva tri procesa u redu spremnih procesa
- ▣ U kritičnu sekciju ulazi P1
- ▣ Nakon izlaska P1 iz kritične sekcije, iz reda spremnih procesa bira se P3
- ▣ Nakon izlaska P3 iz kritične sekcije, iz reda spremnih procesa bira se P1 i tako u krug
- ▣ P2 nikad ne dobija procesor, iako je spreman

# Zahtevi za međusobnu isključivost

- Mora se sprovesti ako u jednom trenutku samo jedan proces sme da koristi resurs
- Ne sme se desiti uzajamno blokiranje ili gladovanje
  - ▣ Proces ne sme beskonačno dugo da čeka na ulazak u kritičnu sekciju
- U kritičnu sekciju se ulazi bez odlaganja ukoliko nijedan drugi proces nije u sekciji
- Ne mogu se praviti pretpostavke o relativnim brzinama procesa
- Proces mora izaći iz kritične sekcije u konačnom vremenu

# Sinhronizacija

- Uvođenje kritične sekcije može da obezbedi sekvencijalan pristup resursima
- Za sinhronizaciju rada procesa putem signalizacije koriste se mehanizmi za sinhronizaciju
- Primeri/Konkurentnost/BaferNesinhronizovano

# Sinhronizacija

- Koncepti višeg nivoa za sinhronizaciju
  - ▣ Semafori
  - ▣ Uslovne promenljive
  - ▣ Monitori



# Semafori

- Dizajnirao ih Edsger Dijkstra (1960-ih) kao deo OS
- Semafor je deljeni brojač (celobrojna vrednost)
- `semWait()` ili `P()` ili `down()`
  - ▣ Umanji brojač za 1
  - ▣ Sačekaj da brojač bude veći ili jednak nuli
- `semSignal()` ili `V()` ili `up()`
  - ▣ Povećaj brojač za 1
- Obe operacije su atomične

# Opis operacija semafora

```
class Semaphore {  
    private:  
        int count;  
        queue q; //red cekanja  
  
    public:  
        Semaphore(int initCounter): count(initCounter) {}  
        void semWait();  
        void semSignal();  
};
```

# Opis operacija semafora

```
void Semaphore::semWait() {  
    count--;  
    if (count < 0) {  
        //dodaj proces u red cekanja q  
        //prebaci proces u stanje blokiran  
    }  
}  
  
void Semaphore::semSignal() {  
    count++;  
    if (count <= 0) {  
        //izbaci proces iz reda cekanja q  
        //prebaci proces u stanje spreman  
    }  
}
```

# Brojač semafora

- Vrednost brojača označava
  - ▣ Ako je **count**  $\geq 0$ 
    - broj procesa koji mogu izvršiti operaciju a da ne budu blokirani
  - ▣ Ako je **count**  $\leq 0$ 
    - broj blokiranih procesa koji čekaju u redu

# Varijante semafora

- Prema tipu brojača
  - ▣ Brojački semafor ili opšti semafor
    - Opisana varijanta u kojoj brojač može dobiti proizvoljnu celobrojnu vrednost
  - ▣ Binarni semafor
    - Specijalna varijanta semafora u kojoj brojač može imati vrednosti 0 ili 1
    - Praktično se ponaša kao muteks

# Varijante semafora

## □ Prema načinu raspoređivanja

### ▣ Jak semafor

- proces koji je prvi otišao u čekanje, prvi biva signaliziran i postaje spreman
- procesi na čekanju se uvezuju u FIFO red
- garantuju da nema gladovanja procesa

### ▣ Slab semafor

- nije određeno koji od blokiranih procesa će postati spreman

# Primer upotrebe semafora

---

- Primeri/Konkurentnost/BaferSemafor

# Semafori problemi

- Algoritmi često zahtevaju upotrebu više semafora
- Pozivi semSignal i semWait nisu upareni
- Potrebno je pratiti sve semSignal i semWait pozive
- Neispravan rad ako redosled ovih operacija nije odgovarajući
- Teško je pronaći takve greške u programu
- Semafori se koriste istovremeno za obezbeđivanje međusobne isključivosti i sinhronizacije, što bi trebalo biti različito tretirano



# Uslovne promenljive

- Uslovna promenljiva predstavlja uslov
  - ▣ Na koji nit može da čeka da se ispuni
  - ▣ Za koji nit može da obavesti druge niti da je ispunjen
- Veoma koristan mehanizam za sinhronizaciju putem signalizacije između niti
- Operacije
  - ▣ wait
    - nit se blokira dok se uslov ne ispuni
  - ▣ signal
    - obaveštenje **jednoj** niti, koja čeka na uslov, da je uslov ispunjen
  - ▣ broadcast
    - Obaveštenje **svim** nitima, koje čekaju na uslov, da je uslov ispunjen

# Uslovne promenljive u C++11 standardu

- Predstavljene klasom `condition_variable`
- Metode se moraju pozivati unutar kritične sekcije zaštićene `unique_lock` objektom

# condition\_variable metode

- `wait(unique_lock<mutex> l)`
  - ▣ Nit se blokira dok neka druga nit nad ovim `condition_variable` objektom ne pozove `notify()` ili `notify_all()`
  - ▣ Otključava propusnicu pre blokiranja
  - ▣ Čeka na propusnicu da bi izašla iz `wait`
  - ▣ Nakon izlaska iz `wait`, mora se ponovo proveriti uslov jer je uslov možda promenjen dok je nit dobila propusnicu
  - ▣ Zato `wait` uvek ide unutar `while` petlje!
  - ▣ Lažno buđenje (*spurious wakeup*)
    - Nit može izaći iz čekanja čak i ako nije notifikirana!

# Condition variable metode

- notify\_one()
  - ▣ Kao signal
  - ▣ U stanje spreman prevodi **jednu od niti** koje su pozvale wait nad ovim condition\_variable objektom
- notify\_all()
  - ▣ Kao broadcast
  - ▣ U stanje spreman prevodi **sve niti** koje su pozvale wait nad ovim condition\_variable objektom

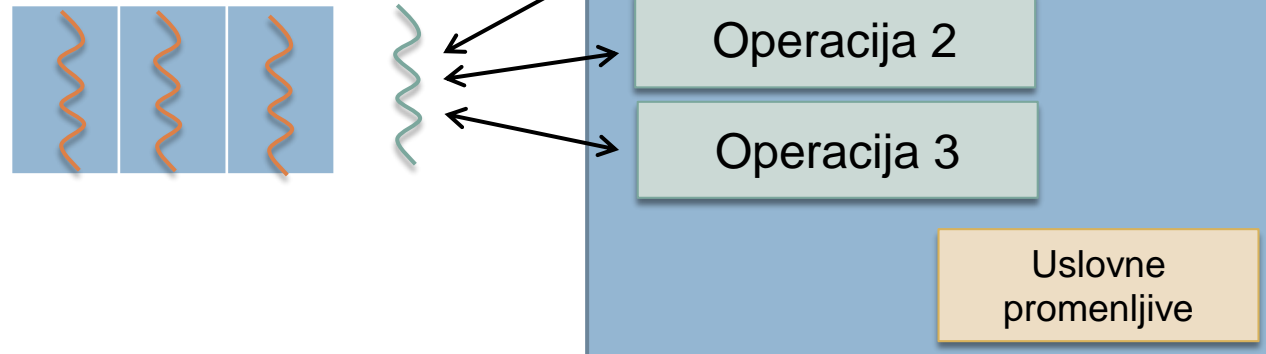
# Primer – uslovne promenljive

---

- Primeri/Konkurentnost/BaferUslovnaPromenljiva

# Monitori

- Softverski patern koji enkapsulira
  - ▣ Podatke
  - ▣ Propusnicu
  - ▣ Operacije nad podacima
  - ▣ Uslovne promenljive za sinhronizaciju



# Monitori

- Procesi „vide“ monitor kao *black-box*
- Pristupaju podacima putem metoda
- Metode se jednim delom izvršavaju sekvencijalno jer je potrebno zauzeti propusnicu
- Metode implementiraju internu logiku sinhronizacije
  - ▣ Mogu blokirati nit
  - ▣ Niti ne moraju da vode računa o sinhronizaciji, monitor garantuje ispravan rad metoda

# Monitori – semantika signaliziranja

- Operacija `signal()` može da ima dva različita značenja
- Hoare monitori (1974)
  - ▣ Operacija `signal` **odmah** aktivira signaliziranu nit
  - ▣ Nit koja je izvršila signalizaciju se blokira
- Mesa monitori (1980)
  - ▣ Operacija `signal` samo prevodi signaliziranu nit u stanje Spreмна
  - ▣ Nit koja je izvršila signalizaciju nastavlja da se izvršava
  - ▣ Signalizirana nit mora da sačeka propusnicu da bi krenula da radi
  - ▣ Signalizirana nit mora ponovo da proverí uslov kad se aktivira, jer je moglo doći do promene uslova dok je čekala na propusnicu
  - ▣ U C++11 metoda `notify()` klase `condition_variable` radi na ovaj način



# Monitori-primer

---

- Primeri/Konkurentnost/BaferMonitor