Конкурентно програмирање

Зоран Јовановић

Садржај

- Основни појмови
- Семафори
- Региони
- Монитори
- Дељене променљиве

Основни појмови

- *Секвенцијално извршавање* је такво извршавање где се следећа рачунарска операција или програмска наредба извршава тек након што је претходна завршена, у складу са редоследом који је задат програмом.
- *Секвенцијални ӣроӣрам* је програм код кога постоји само један секвенцијалан ток извршавања у времену. То значи, без обзира на каквој рачунарској платформи се програм извршава, да ће се, када су улазни параметри програма исти, увек извршавати исте наредбе, истим јединственим редоследом који је задат изворним програмом.

Основни појмови

- *Паралелно извршавање* означава истовремено извршавање више рачунарских операција, секвенци операција, програма или делова једног програма.
- *Паралелни ӣроӣрам* је сваки програм који користи посебну синтаксу за означавање делова програмског кода који се могу извршавати паралелно.
- *Паралелни рачунарски сисӣем* је сваки рачунарски систем који је у стању да истовремено извршава два или више делова једног (или више) програма.

Основни појмови

- *Ток извршавања програма (execution flow)*, *програмски ток (program flow), инструкцијски ток (instruction flow)* или *нит тока контроле* (*thread of control*), како се још зове, је назив за скуп наредби програма које се извршавају одређеним секвенцијалним редоследом.

- *Конкурентно извршавање* подразумева извршавање више програмских токова једног програма тако да они напредују у времену (бар два од њих), али се они не морају обавезно извршавати истовремено (што би био случај код паралелног извршавања).

Основни појмови

- Секвенцијални програм – један ток контроле
- Конкурентан програм – више токова контроле

- Програми комуницирају преко:
  - дељених променљивих (*shared variables*) или
  - прослеђивања порука (*message passing*)

Типови машина

- Системи са дељеном меморијом (*Shared memory)* – мултипроцесори
- Системи са дистрибуираном меморијом (*Distributed memory*) – мултикомпјутери

- Мреже – *WAN* и *LAN*
- Бежичне мреже – *WLAN*, мобилна телефонија

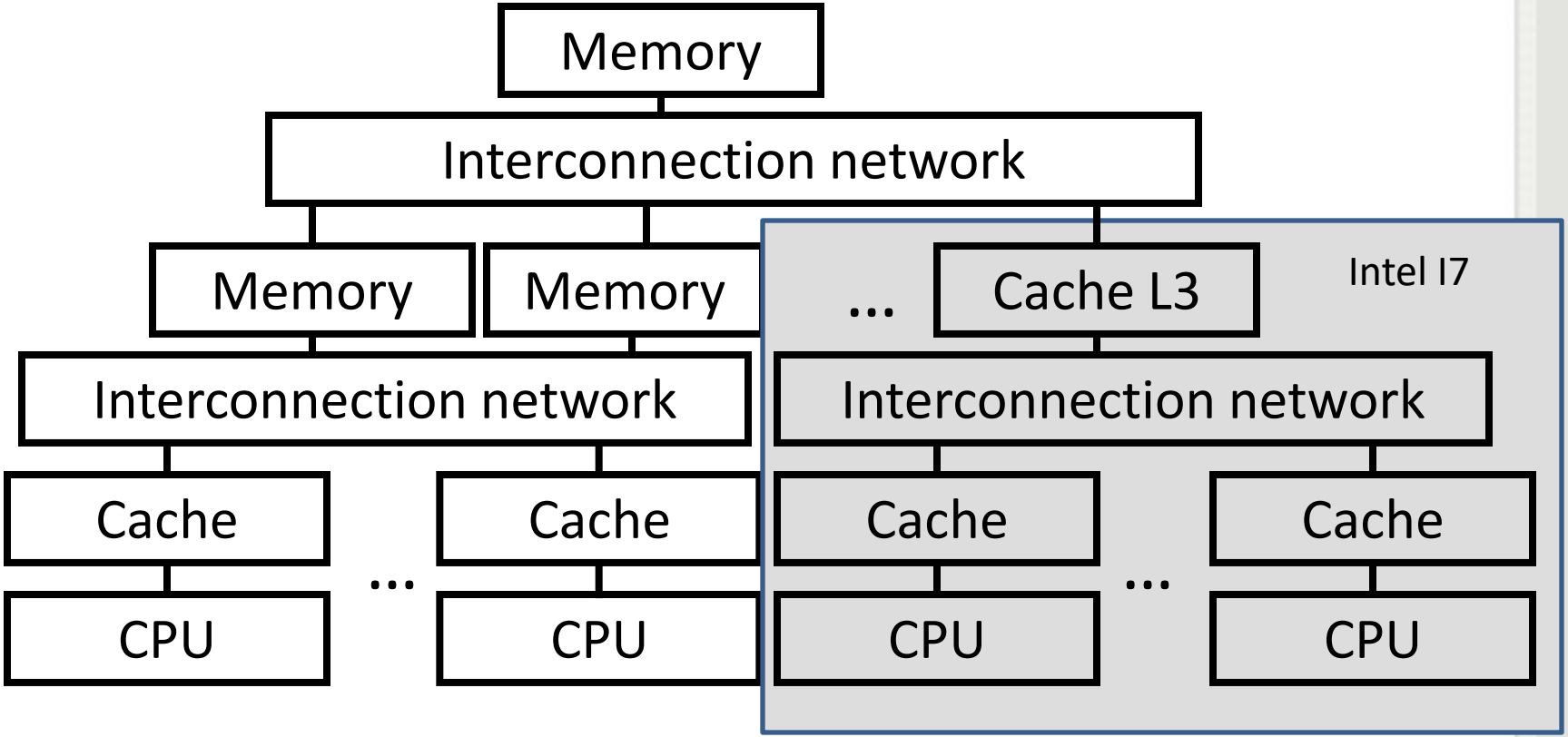
Системи са дељеном меморијом

- Системи са дељеном меморијом (*Shared memory*) мултипроцесори – сваки *CPU* поседује сопствени кеш (неки пут такође и локалну меморију) и меморију коју деле са другим процесорима кроз интерконекциону мрежу

- Типови:
  - *UMA – Uniform Memory Access* (*Symmetric multiprocessing*)
  - *NUMA – No Uniform Memory Access*

- Конзистенција копија у кеш меморијама

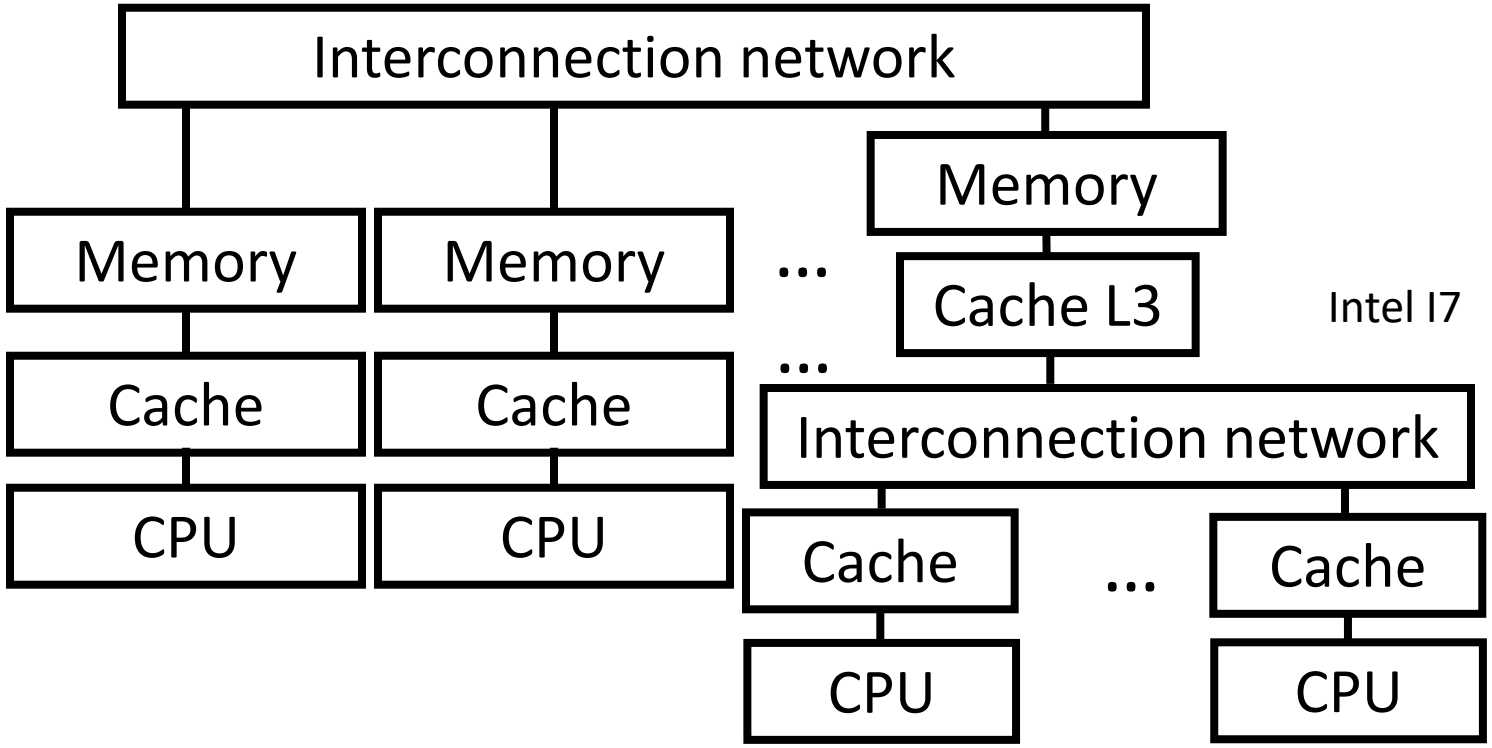
Системи са дељеном меморијом



Системи са дистрибуираном меморијом

- Аутономни процесори који не поседују дељену меморију и који користе комуникационе мреже за прослеђивање порука (*Distributed memory multiprocessors*)
- Типови:
  - ***Tightly coupled –*** дистрибуиран систем са широким пропусним опсегом и малим кашњењем комуникационе мреже, регуларном топологијом и локалном поузданом интерконекционом мрежом. (*Hypercube, Mesh, Star-Graph*).
  - ***Loosely coupled –*** дистрибуирани систем са *LAN* или *WAN* мрежом. Мрежа радних станица, кластер радних станица, интернет. Већа кашњења, мања поузданост веза и рутирање није засновано на унапред познатој топологији.

Системи са дистрибуираном меморијом



Tightly coupled

- Регуларна топологија интерконекционе мреже
- Мали дијаметар интерконекционог графа
- Мала физичка растојања
- Једноставно рутирање – хардверски засновано са уграђеним избегавањем мртвог блокирања
- Дистрибуирано рутирање – одлука о излазном каналу мора се заснивати на заглављу (адреса одредишта, или чак део адресе одредишта)
- Адаптивно (алтернативно) рутирање – примарно због умањене осетљивости на отказе

Loosely coupled

- **LAN** – Локалне мреже
  - Брзине 0.01-100 *Gbps*
  - Без умањене осетљивости на отказе (*bus*, *star* и *ring*)
  - Са умањеном осетљивошћу на отказе - (*token-ring*, *FDDI* и *star-ring*)
  - Хибридне (*HUB*, *switch*)
- **WAN** – Глобалне мреже
  - Брзине 28.8 Kbps – 400 *Gbps*
  - Непоуздани комуникациони канали
  - Различити типови *Dial-Up*, *ADSL*, *Cable*, *Wimax*, *4G*, *fiber*.

Интернет

- Милиони рачунара у мрежи са процесима који раде као конкурентни (дистрибуирани) програми
- Стандардни интерфејси, протоколи – стандарди за конкурентно програмирање
- *Protocol layering* – изоловање интерфејса да би се енкапсулирао софтвер на сваком слоју (нема комуникације или синхронизације између софтвера на различитим слојевима)

Бежичне мреже

- *WLAN* бежични *TCP/IP* – Интернет окружење
- Мобилна телефонија – *SMS* – *Short messages*
- *WAP* – *Wireless application protocol*
- *GPRS*, *3G-5G* – довољан капацитет за мобилне интернет сервисе

Distributed shared memory

- Дистрибуирана имплементација *shared memory* апстракције – софтвер или хардвер
- Виртуелни меморијски адресни простор са страницама дистрибуираним у меморијама *distributed memory machine*
- Реплициране копије и протоколи за конзистенцију страница
- Разлог – креирање *shared memory* апстракције за *shared memory* конкурентне програме у дистрибуираним системима

Класе конкурентних и дистрибуираних апликација

- **Редуковање времена извршавања –** предуслов: мала комуникациона кашњења и паралелизам у апликацији. Различити нивои паралелизма: *coarse grain*, *medium grain* (*loop level*) и *fine grain*. *Tightly coupled* системи за *medium* и *fine grain*.
- ***Fault tolerance***: откази се превазилазе репликацијом функција или података уз аутоматски опоравак од грешке. Дистрибуирани системи могу имати географску дистрибуцију и нема критичног ресурса. Пример: *fly by wire*.
- **Специалне функционалности дистрибуираних система**: филе сервери (пример *Picture Archiving and Communication Systems*), нумерички сервери, производња *IC* и комуникациони сервери.
- **Апликација је дистрибуирана**: резервација авио карата и имеил

Проблем утркивања – *race condition*

```
x = -1; y = -1;
process P1 {
    x = 1;
    y = 2;
}
process P2 {
    x2 = x;
    y2 = y;
}
```

Које су могуће вредности за x2 и y2?

Основни појмови

- Међусобно искључивање (*Mutual exclusion*) – искази који се не извршавају у исто време
- Условна синхронизација (*Conditional synchronization*) – закашњавање све док услов не постане испуњен

Конкурентно програмирање
Критична секција, AMOP, Await

Међусобно искључивање

- Недељиво извршавање комбинације суседних атомских акција (АА) као крупнија недељива акција
- Тај скуп исказа се назива критична секција *(critical section)*. Извршавање критичне секције се мора обавити са међусобним искључивањем. То се мора урадити само за дељене ресурсе (променљиве).

At-Most-Once-Property

- Критична референца – референца на променљиву која се мења од стране другог процеса
- Када додељивање x:= v има *At-Most-Once-Property* (*AMOP*)?

1. Када v садржи највише једну критичну референцу и x се не чита од стране другог процеса или
2. Када v не садржи критичну референцу, и x се чита од стране другог процеса

- Тада => x:=v се јавља као AA

Пример

```
int x:=0, y:=0;
co
    x:=y+1; || y:=y+1;
oc
```

крајња вредност *x* је 1 или 2 – али **да**-*AMOP*
(*y* није критична референца за процес 2)

Јављају се као атомске акције додељивања

```
int x:=0, y:=0;
co
    x:=y+1; || y:=x+1;
oc
```

– **није** *AMOP*

Await исказ

- Ако није испуњен *At-Most-Once-Property* – неопходна је *coarse grained* (крупнија) атомска акција
- Пример операције у вези повезане листе или бафера – уметање или изостављање елемената – за један елемент
- <await(B) S;>
- B – boolean – специфицира *delay condition* – услов кашњења
- S секвенца исказа за које је гарантовано да терминирају

Међусобно искључивање и синхронизација са *await*

- <S;> - међусобно искључивање

- <await(B);> - условна синхронизација
  – B == true тада нема кашњења.
- B – када има *At-most-Once-Property* –
  while not(B) skip; – *condition synchronization* – празно тело петље => ***spin loop***

- **безусловна атомска акција <S;>**
- **условна атомска акција <await(B) S;>**

Произвођач/потрошач синхронизација

- Дељени бафер за копирање низа
- Решење са локацијом за један елемент и две дељене променљиве за бројање
- Уводи се синхронизациони предикат – логички исказ који мора бити *true*
- Синхронизациони предикат: *c* <= *p* <= *c+1*

  p==c слободна локација;

  p==c+1 попуњена локација

- Решење са запосленим чекањем

Програм – копирање низа (1)

```
int buf, p = 0, c = 0; /*deljene promenljive*/
process Producer {
    int a[n];
    while (p < n) {
        <await(p == c);> /* Sinhronizacija na empty*/
        buf = a[p];
        p = p+1;
    }
}
```

Програм – копирање низа (2)

```
process Consumer {
    int b[n];
    while (c < n) {
        <await(p > c);> /* Sinhronizacija na full*/
        b[c] = buf;
        c = c+1;
    }
}
```

Проблем критичне секције

- Више (n) процеса са кодом

```
process CS[i = 1 to n] {
    while (true) {
        entry protocol;
        kritična sekcija;
        exit protocol;
        nije kritična sekcija;
    }
}
```

Особине и лоша стања

- *Mutual exclusion* (**међусобно искључивање**) – највише један процес извршава своју критичну секцију
- Не сме да постоји ***Deadlock*** **(*Livelock*)** – бар један процес ће успети
- **Нема непотребног кашњења** – ако други нису у критичној секцији, и један процес чека – неће бити закашњен
- ***Eventual entry*** – на крају ће сваки процес који жели, успети да уђе у критичну секцију

Решења – два процеса

- < > за све критичне секције са *await*
- ¬ *(in1 and in2)* предикат за два процеса
- Два процеса –
  први:

  <await(!in2) in1 = true;> /* entry */
  kritična sekcija;
  in1 = false; (*At-Most-Once-Property*)

Нема непотребног кашњења

Критична секција – два процеса (1)

```
bool in1 = false, in2 = false;
process CS1 {
    while (true) {
        <await(!in2) in1 = true;>      /* entry */
        kritična sekcija;
        in1 = false;                   /* exit */
        nije kritična sekcija;
    }
}
```

Критична секција – два процеса (2)

```
process CS2 {
    while (true) {
        <await(!in1) in2 = true;>      /* entry */
        kritična sekcija;
        in2 = false;                   /* exit */
        nije kritična sekcija;
    }
}
```

Критична секција – *lock*

- једна променљива за два стања
- *lock == (in1 or in2 or ....)*
- Ако је *lock = false* – нико није у критичној секцији
- Ако је *lock = true* – постоји процес у критичној секцији
- симетрично решење => за више процеса

Решење

```
bool lock = false;
process CSi {
    while (true) {
        <await(!lock) lock = true;>  /* entry */
        Критична секција;
        lock = false;                /* exit */
        није критична секција;
    }
}
```

Test and Set

Специјална инструкција за процесоре

Атомска акција – узме стару *boolean* вредност и враћа је, уједно недељиво поставља вредност на *true*

```
bool TS(bool lock) {
    < bool initial = lock;        /* sačuvaj vrednost */
    lock = true;                  /* postavi true */
    return initial; > }           /* vrati inicijalnu vrednost */
```

Test and Set – Критична секција

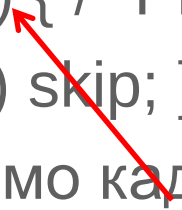
```
bool lock = false;
process CSi {
    while (true) {/*entry spin lock ⇔ while lock skip;*/
        while (TS(lock)) skip;          /* i lock = true; kao AA,
    kada se izlazi iz petlje (ulazi u kritičnu sekciju)*/
        kritična sekcija;
        lock = false;                   /* exit */
        nije kritična sekcija;
    }
}
```

Test and Set – Проблем

- *spin locks* – загушење меморије и кеш меморија
- Упис у *lock* изазива инвалидацију кеш меморија кеш меморија свих процесора са дељеном меморијом
- Измена *entry* протокола која повећава вероватноћу успеха, а не уписује стално

```
while (lock) skip;  /* spin dok je lock true */
while (TS(lock)) { /* Probaj da prvi uzmeš lock */
        while (lock) skip; } /* spin ponovo ako nisi uspeo */
```

Инвалидације само када неко улази у критичну секцију



Test and Test and Set

Синхронизациони алгоритми

Tie-Breaker (Петерсонов) алгоритам

- Додатна променљива која индицира ко је задњи у критичној секцији - уједно *fairness*
- једноставне променљиве и секвенцијални искази

- *Entry* протоколи који **не раде**:
  in1 и *in2* иницијално *false*

```
P1:     while (in2) skip;
        in1 = true;
P2:     while (in1) skip;
        in2 = true;
```

Није испуњено међусобно искључивање због АА!

Tie-Breaker – coarse grain (1)

```
bool in1 = false, in2 = false;
int last =1;
process CS1 {
    while (true) {
        in1 = true; last = 1;      /* entry protokol */
        <await(!in2 or last ==2);>
        kritična sekcija;
        in1 = false;               /* exit protokol */
        nije kritična sekcija;
    }
}
```

Tie-Breaker – coarse grain (2)

```
process CS2 {
    while (true) {
        in2 = true; last =2;       /* entry protokol */
        <await(!in1 or last ==1);>
        kritična sekcija;
        in2 = false;               /* exit protokol */
        nije kritična sekcija;
    }
}
```

Tie-Breaker – fine-grain решење

```
bool in1 = false, in2 = false;
int last =1;
process CS1 {
    while (true) {
        in1 = true; last = 1;      /* entry protokol */
        while (in2 and last == 1) skip;
        kritična sekcija;
        in1 = false;               /* exit protokol */
        nije kritična sekcija
    }
}
```

Tie-Breaker – fine-grain решење

```
process CS2 {
    while (true) {
        in2 = true; last = 2;      /* entry protokol */
        while (in1 and last ==2) skip;
        kritična sekcija;
        in2 = false;               /* exit protokol */
        nije kritična sekcija
    }
}
```

Објашњење (1)

- <await(B)> ⇔ while (not B) ако је B AMOP
- ! (!in2 or last == 2) ⇔ in2 and !(last == 2)

- last је или 1 или 2 => *!(last == 2) ⇔ last == 1*
- *in2 and last* == 1 да ли поседује *AMOP* ?
  Не (и *in2* и last су критичне референце)

Али: Ако је *in2 false* за *CS1* и пре испитивања *last == 1*, *in2* постане *true* јер *CS2* управо изврши *in2 = true*, да ли могу истовремено у критичну секцију?

Објашњење (2)

Исход 1 – процес *CS2* први изврши *last = 2*; тада мора да чека у петљи, јер је *in1 and last == 2* сада true (*CS2* је дао приоритет CS1 за истовремено *in1* и *in2 true*). Промена *in2* није утицала на улазак, јер *CS2* даје приоритет *CS1*.

Исход 2 – процес *CS1* испитује *last == 1*, али је *in2* већ испитао и било је *false* => улази у критичну секцију. =>

Процес који је први променио *last* улази у *CS* ако су и *in1* и *in2* постали *true*. (у претходном случају *CS1*). Упис у ласт је АА.

Свеукупно, није *AMOP*, али се јавља као *AMOP*

Tie-Breaker за *n* процеса

- n stanja – који процес прелази у следеће стање одређује *Tie-Breaker* (Петерсонов) алгоритам за два процеса
- Највише један у једном тренутку може да прође свих *n-1* стања
- Уводе се два *integer* низа *in[1:n]*, *last[1:n]*
- *in[i]* – кроз које стање процес CS[i] пролази
- *last[j]* – који је процес последњи започео (ушао у) стање *j*

Tie-Breaker за *n* процеса

- спољна петља n-1 пута
- унутрашња петља – *CS[i]* чека у достигнутом стању ако постоји процес у вишем или истом стању и CS[i] је био последњи који је ушао у стање *j*
- ако нема процеса у вишем стању који чека на улазак у критичну секцију или други процес улази у стање *j* - *CSi* може да пређе у следеће стање
- Највише *n-1* процеса *CSi* може да прође прво стање, *n-2* процеса *CSi* друго, ..., један процес *CSi* последње стање – Критична секција

Филтрирање – *Filter algorithm*

Tie-Breaker за *n* процеса

```
int in[1:n] = ([n] 0), last[1:n] = ([n] 0);
process CS[i = 1 to n] {
    while (true) {/* entry protokol */
        for [j = 1 to n]  {
            in[i] = j; last[j] = i;
            for [k = 1 to n st i != k] {
                while (in[k] >= in[i] and last[j] == i) skip;
            }
        }
        kritična sekcija;
        in[i] = 0;
        nije kritična sekcija
    }
}
```

Објашњење (1)

- Спољна петља – пролазак кроз стања – иницијално у прво стање и маркирање да је *CSi* последњи који је ушао у прво стање
- Унутрашња петља проверава са свим другим процесима да ли постоји процес у истом или вишем стању. Ако се наиђе на процес у истом или вишем стању, додатно се испитује да ли постоји други процес који је после *CSi* ушао у текуће стање процеса *CSi*
- Ако се то десило, *while* услов је 0, за било које k и *CSi* прелази у више стање

Објашњење (2)

- Последњи који је ушао у неко стање изгура све остале процесе у том стању у више стање.
- Један процес стиже до највишег стања и за тај процес *CSi* не постоји *in[k] >= in[i]* => типично (али не гарантовано) пролази све итерације унутрашње и спољашње петље
- Дакле, типично се изврше све итерације унутрашње петље за све преостале више вредности j, притом означавајући да је био задњи процес који је ушао у стање j.

Ticket алгоритам

- Сви који покушавају да уђу у критичну секцију прво добију *ticket* са бројем у редоследу доласка
- Правична (*fair*) критична секција
- Један по један у редоследу доласка (*ticket*)
- Ресторански сервиси – *fair* сервиси

Ticket алгоритам – *coarse grain*

```
int number = 1, next = 1, turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
    while (true) {
        < turn[i] = number; number = number +1;>
        <await(turn[i] = = next);>
        critical section;
        <next = next +1;>
        noncritical section
    }
}
```

Fetch and add

- Специјална инструкција за процесоре
- Инкрементирање променљиве са константом као атомска акција уз враћање старе вредности

FA(var, incr):

<int tmp = var; var = var + incr; return(tmp);>

- употреба FA за <turn[i]=number; number=number+1;>
- остало има *AMOP* јер сваки процес има свој *turn[i]*, а инкрементирање *next* је на крају критичне секције – само један процес га модификује у једном тренутку!

Ticket алгоритам – *fine grain*

```
int number = 1, next = 1, turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
    while (true) {
        turn[i] = FA(number,1);        /* entry protocol */
        while (turn[i] != next) skip;
        kritična sekcija;
        next = next +1;                /* exit protocol */
        nije kritična sekcija
    }
}
```

Bakery алгоритам

- Личи на *ticket* алгоритам без *next*. Извлачи се број већи од било ког другог процеса интеракцијом са њима
- Узајамна провера између процеса
  (ко је последњи?)
- Узима се за један већа вредност од било које друге у *turn[i]*

Bakery – coarse grain

```
int turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
    while (true) {
        < turn[i] = max (turn[1:n]) + 1;>
        for [j = 1 to n st j != i]
            <await(turn[j] == 0 or turn[i] < turn[j]);>
        kritična sekcija;
        turn[i] = 0;
        nije kritična sekcija
    }
}
```

Bakery 2 – тражење *max fine grain*

- Два процеса прво
- **Није решење 1**: иницијално *turn1* и *turn2* су 0

```
turn1 = turn2 + 1;        /* entry 1 */
while (turn2 != 0 and turn1 > turn2) skip;

turn2 = turn1 + 1;        /* entry 2 */
while (turn1 != 0 and turn2 > turn1) skip;
```

- *turn1* и *turn2* могу оба да постану 1 и оба да уђу у критичну секцију јер су *AA load*, *add*, *store*

Bakery 2 – тражење *max fine grain*

- Два процеса
- **Није решење 2:** иницијално *turn1* и *turn2* су 0

```
turn1 = turn2 + 1;        /* entry 1 CS1*/
while (turn2 != 0 and turn1 > turn2) skip;

turn2 = turn1 + 1;        /* entry 2 CS2*/
while (turn1 != 0 and turn2 >= turn1) skip;
```

- CS1 чита *turn2* (0) и онда *CS2* прође све кораке и уђе у *CS* јер је *turn1* још једнако 0
- *turn1* тада постаје 1 и *CS1* улази у *CS*
- Приоритет дат *CS1* нема ефекта за овај редослед АА

Bakery 2 – тражење *max fine grain*

- Два процеса коректно

turn1 = 1; turn1 = turn2 + 1; /* entry 1 CS1*/
while (turn2 != 0 and turn1 **>** turn2) skip;
turn2 = 1; turn2 = turn1 + 1; /* entry 2 CS2*/
while (turn1 != 0 and turn2 **>=** turn1) skip;

- За иницијалне вредности *turn* различите од 0, OK!
- Није симетрично решење – када би се нашло симетрично решење – може се проширити на n процеса за тражење max од n

Симетрично решење за 2 процеса

- Уведимо уређен пар: *(a,b) > (c,d) == true* акко *a > c* или *a == c* и *b > d* и *false* у свим другим случајевима
- turn1 > turn2 ⇔ (turn1,1) > (turn2,2)
- turn2 **>=** turn1 ⇔ (turn2,2) **>** (turn1,1)
- Ако *turn2 == turn1* онда 2 > 1
- Симетрично решење *(turn[i],i) > (turn[j],j)*
- Процес *CS[i]* – Поставља свој turn на 1, тражење max од *turn[j]*, и додавање 1.

Bakery n – налажење *max fine grain*

```
int turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
    while (true) {
        turn[i] = 1; turn[i] = max (turn[1:n]) + 1;
        for [j = 1 to n st j != i]
            while (turn[j] != 0 and (turn[i],i) > (turn[j],j)) skip;
        kritična sekcija;
        turn [i] = 0;
        nije kritična sekcija
    }
}
```

Bakery – коментари

- Два или више процеса могу да имају исту вредност *turn* – као код два процеса !!!
- Унутрашња *while* петља уводи редослед чак и када два или више процеса имају исто *turn*
- Атомске акције су исте као и у случају два процеса, ***max* је апроксимирано!**

Апроксимациони алгоритми

- Алгоритми са сукцесивним апроксимацијама за низове, исто израчунавање за све елементе низа, засновано на елементима низа из претходне апроксимације
- Итеративни алгоритми са зависностима по подацима пренетим петљом
- Свака апроксимација – процеси који су *cobegin* искази – паралелно извршавање

Андерсенов алгоритам – *coarse grain*

```
int slot = 0, flag[1:n] = ([n] false); flag[1] = true;
process CS[i = 1 to n] {
    int myslot;
    while (true) {
        <myslot = slot mod n + 1; slot = slot +1;>
        <await(flag[myslot])>
        critical section;
        <flag[myslot] = false;
        flag[myslot mod n + 1)] = true;>
        noncritical section
    }
}
```

Андерсенов алгоритам – *fine grain*

```
int slot = 0, flag[1:n] = ([n] false); flag[1] = true;
process CS[i = 1 to n] {
    int myslot;
    while (true) {
        myslot = FA(slot,1) mod n + 1; /*entry protocol*/
        while (!flag[myslot]) skip;
        critical section;
        flag[myslot] = false; /* exit protocol */
        flag[myslot mod n + 1)] = true;
        noncritical section
    }
}
```

CLH алгоритам – *coarse grain*

```
Node tail = (false);
process CS[i = 1 to n] {
    while (true) {
        Node prev, node = (true);/*entry protocol*/
        <prev = tail; tail = node;>
        <await(!prev.locked);>
        critical section;
        <node.locked = false;> /* exit protocol */
        noncritical section
    }
}
```

Craig, Landin и Hagersten

Get and Set

- Специјална инструкција за процесоре
- Дохвата стару вредност променљиве и поставља нову вредност променљиве као атомску акцију
- GS(var, new):

  <int tmp = var; var = new; return(tmp);>
- Употреба за <prev = tail; tail = node;>

CLH алгоритам – *fine grain*

```
Node tail = (false);
process CS[i = 1 to n] {
    while (true) {
        Node prev, node = (true);/*entry protocol*/
        prev = GS(tail, node);
        while(prev.locked) skip;
        critical section;
        node.locked = false; /* exit protocol */
        noncritical section
    }
}
```

Синхронизација на баријери – Неефикасан приступ

```
while (true) {
    co [i=1 to n]
        Kod za task i;
    oc
}
```

Co започиње креирање *n* процеса у свакој итерацији - неефикасно

Идеја: креирати n процеса на почетку и онда само синхронизовати на крају итерације

Синхронизација на баријери - *Barrier synchronization*

```
process Worker[i = 1 to n] {
    while (true) {
        kod kojim se implementira task i;
        čekanje na svih n taskova da se završe;
    }
}
```

- Баријере су најчешће на крајевима итерације, али могу чак и код међустања
- Једна идеја: дељени бројач који броји до n, инкрементирају га сви процеси, а када се достигне n, онда сви пролазе баријеру

Shared counter barrier

<count = count + 1;>
<await(count == n);>

Ca Fetch and add - FA

```
FA(count,1);
while (count != n) skip;
```

- Проблеми
  - ресет бројача после свих и мора се проверити пре почетка инкрементирања у наредној итерацији;
  - *up/down counter* као решење;
  - проблем ажурирања кеш меморија;

Flags and coordinators

<await(arrive[1] + … + arrive[n] == n);>

Ако се сума израчунава у сваком процесу, опет загушење у кеш меморији

Continue array – Координатор чита *arrive* низ и поставља *continue* низ да се избегне загушење

for [i = 1 to n] <await(arrive[i] == 1);>

for [i = 1 to n] continue[i] =1; /* ko radi reset ? */

Flag синхронизација

- Правила за коректно извршавање:

Процес који чека на *flag* да буде постављен на 1 треба да га врати на 0

Flag треба да буде постављен само ако је враћен на 0

- Радни процес брише *continue*, а *Coordinator* брише *arrive*
- *Coordinator* треба да обрише (врати на 0) *arrive* пре постављања *continue*

Синхронизација на баријери са процесом координатором (1)

```
int arrive[1:n] = ([n] 0); continue[1:n] = ([n] 0);
process Worker[i = 1 to n] {
    while (true) {
        kod taska i;
        arrive[i] = 1;
        <await(continue[i] == 1);>
        continue[i] =0;
    }
}
```

Синхронизација на баријери са процесом координатором (2)

```
process Coordinator {
    while (true) {
        for [i = 1 to n] {
            <await(arrive[i] == 1);>
            arrive[i] = 0;
        }
        for [i = 1 to n] continue[i] = 1;
    }
}
```

Проблеми са запосленим чекањем

- Комплексни алгоритми
- Нема разлике између променљивих намењених синхронизацији и рачунању
- Неефикасно код *multithreading* и код мултипроцесора, ако је број процеса > од броја процесора
- Јавља се потреба за специјалним алатима за синхронизацију - Семафори – аналогија са железничким семафорима – критична секције

Семафори

Семафорске операције

- Декларација sem s; sem lock =1;
- Ако нема декларације иницијалних вредности, => иницијализовано на 0
- Мења се само операцијама wait(s) и signal(s)
- Wait(s): <await(s>0) s = s - 1;>
- signal(s): < s = s + 1; >
- Бинарни семафор: многи процеси покушавају wait(s), али само један може да прође до следећег signal(s)
- Генерални семафор: било која позитивна вредност или 0
- Ако више процеса чека = > типично буђење је у редоследу у коме су били закашњени

Проблем критичне секције

```
sem mutex=1;
process CS[i = 1 to n] {
    while (true) {
        wait(mutex);
        критична sekcija;
        signal(mutex);
        nije kritična sekcija;
    }
}
```

Међусобно искључивање

Producers and consumers (n)

```
process Producer[i = 1 to M] {
    while (true) { …
        /* proizvedi podatak i unesi */
        buf = data;
    …}
}
process Consumer[j = 1 to N] {
    while (true) { …
        /* uzmi rezultat */
        result = buf;
    …}
}
```

Условна синхронизација

Producers and consumers

- *Split binary semaphore* (расподељени бинарни семафор) - Више процеса, једна локација
- *wait* на неком семафору мора да буде улаз
- Два бинарна семафора: *empty* и *full* – као један семафор за критичну секцију расподељен у два семафора
- На сваком трагу извршавања, операција *wait* мора да буде праћена са *signal* на крају дела кода критичне секције. Један семафор (*empty*) је иницијално 1
- Када год постоји неки (један) процес између *wait* и *signal* => сви семафори расподељеног бинарног семафора су 0

Producers and consumers (n)–расподељени семафори

```
typeT buf;
sem empty =1, full = 0;

process Producer[i = 1 to M] {
    while (true) { …
        /* proizvedi podatak i unesi */
        wait(empty);
        buf = data;
        signal(full);
        …
    }
}
```

Producers and consumers (n)–расподељени семафори

```
process Consumer[j = 1 to N] {
    while (true) { …
        /* uzmi rezultat */
        wait(full);
        result = buf;
        signal(empty);
        …
    }
}
```

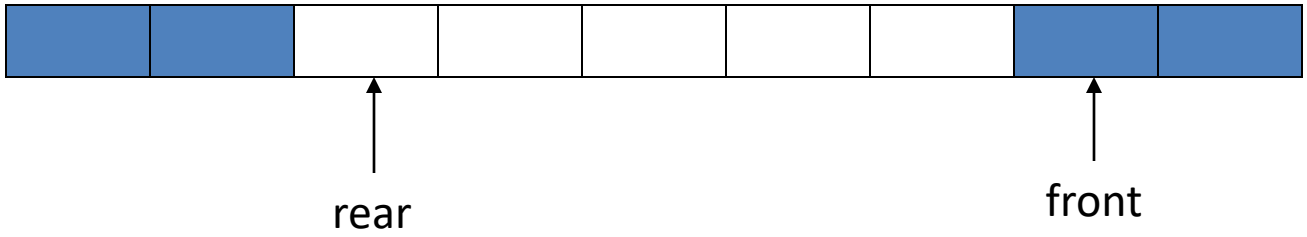
Bounded buffer

Deposit:

buf[rear] = data; rear = (rear + 1) % n;

Fetch:

Result = buf[front]; front = (front + 1) % n;



Bounded buffer по један произвођач и потрошач

```
typeT buf[n];
int front = 0, rear = 0;
sem empty = n, full = 0;
process Producer {
    while (true) { …
        /* proizvedi podatak */
        wait(empty);
        buf[rear] = data; rear = (rear + 1) % n;
        signal(full);
        …
    }
}
```

Bounded buffer по један произвођач и потрошач

```
process Consumer {
    while (true) { …
        /* dohvati rezultat */
        wait(full);
        result = buf[front]; front = (front + 1) % n;
        signal(empty);
        …
    }
}
```

Bounded buffer M произвођача и N потрошача

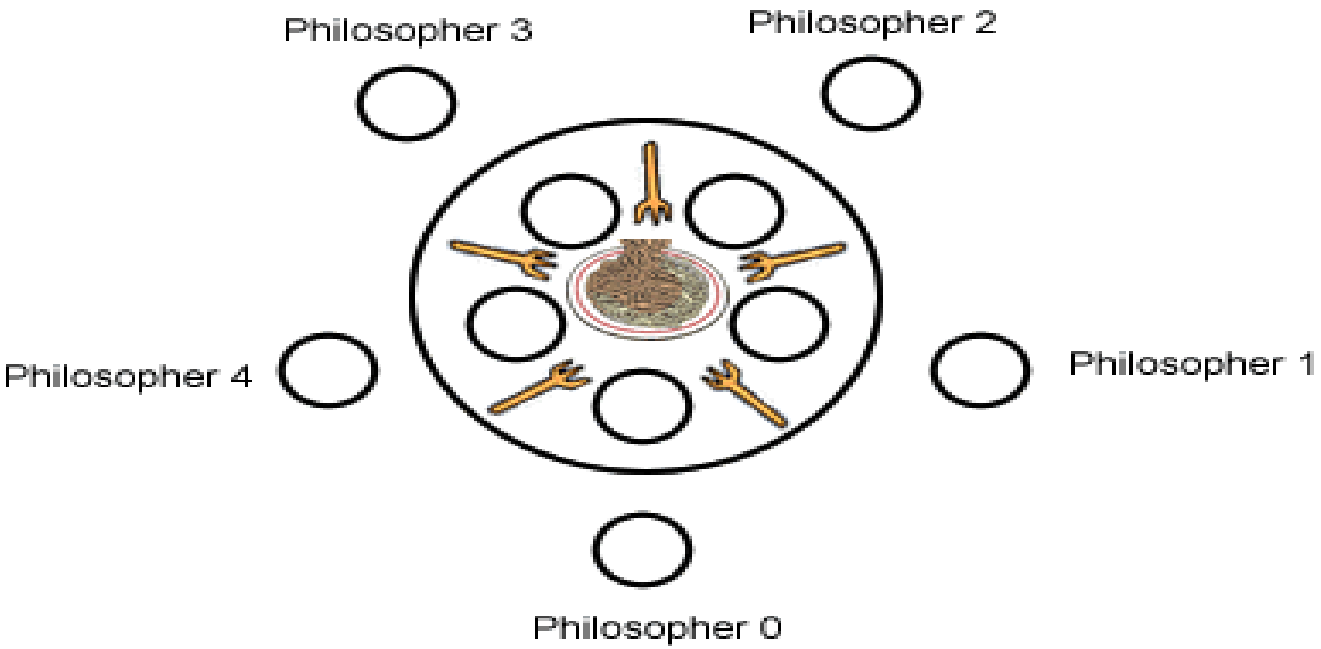
```
typeT buf[n];
int front = 0, rear = 0;
sem empty = n, full = 0;
sem mutexD = 1, mutexF = 1;

process Producer[i = 1 to M] {
    while (true) { …
        /* proizvedi podatke */
        wait(empty); wait(mutexD);
        buf[rear] = data; rear = (rear + 1) % n;
        signal(mutexD); signal(full);
        …
    }
}
```

Bounded buffer M произвођача и N потрошача

```
process Consumer[j = 1 to N] {
    while (true) { …
        /* dohvati resultat i potroši */
        wait(full); wait(mutexF);
        result = buf[front]; front = (front + 1) % n;
        signal(mutexF); signal(empty);
        …
    }
}
```

Dinning philosophers



Dinning philosophers

```
process Philosopher[i = 0 to 4] {
    while (true) {
        razmišljaj;
        dohvati viljuške;
        jedi;
        oslobodi viljuške;
    }
}
```

Dinning philosophers

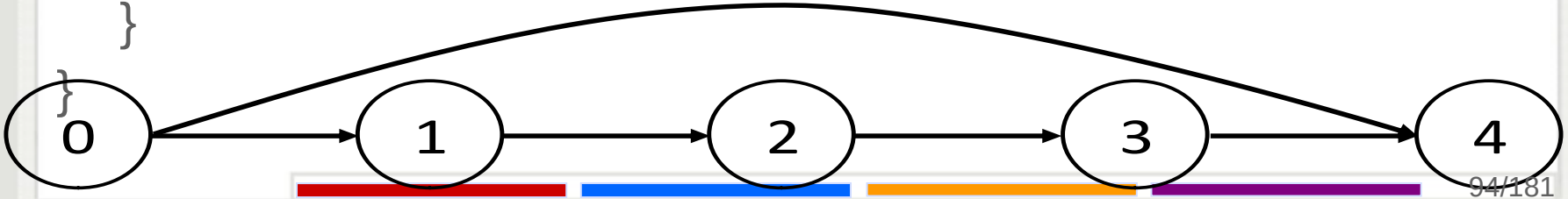
- Виљушке су дељене – критичне секције
- **Мртво блокирање** (***Deadlock***) ако сви процеси имају исти редослед дохватања (нпр. лева па десна) при коме долази до кружног чекање (затворен пут захтева за ресурсима)

Мртво блокирање (*Deadlock*)

- *Deadlock* настаје ако су задовољена следећа четири услова:
  - Процеси имају право ексклузивног приступа ресурсима (*Mutual exclusion*).
  - Процес држи ресурс све време док чека приступ другом ресурсу (*Wait and hold*).
  - Процес не може преузети ресурс на који чека (*No preemption*).
  - Постоји затворен круг процеса који чекају један другог - један процес чека на испуњење услова који обезбеђује други процес, други процес чека на испуњење услова који обезбеђује трећи процес итд., а последњи процес у том низу чека на испуњење услова који обезбеђује први процес (*Circular wait*).

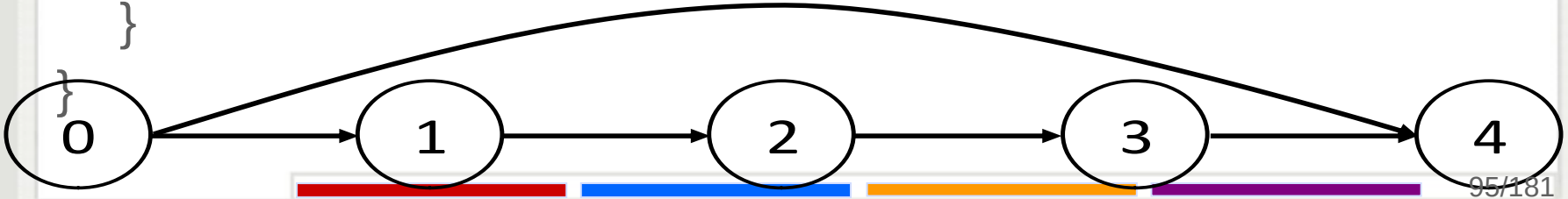
Dinning philosophers

```
sem fork[5] = {1,1,1,1,1};
process Philosopher[i = 0 to 3] {
    while (true) {
        /* leva pa desna */
        wait(fork[i]); wait[fork[i+1];
        eat;
        signal(fork[i]); signal(fork[i+1]);
        think;
    }
}
```



Dinning philosophers

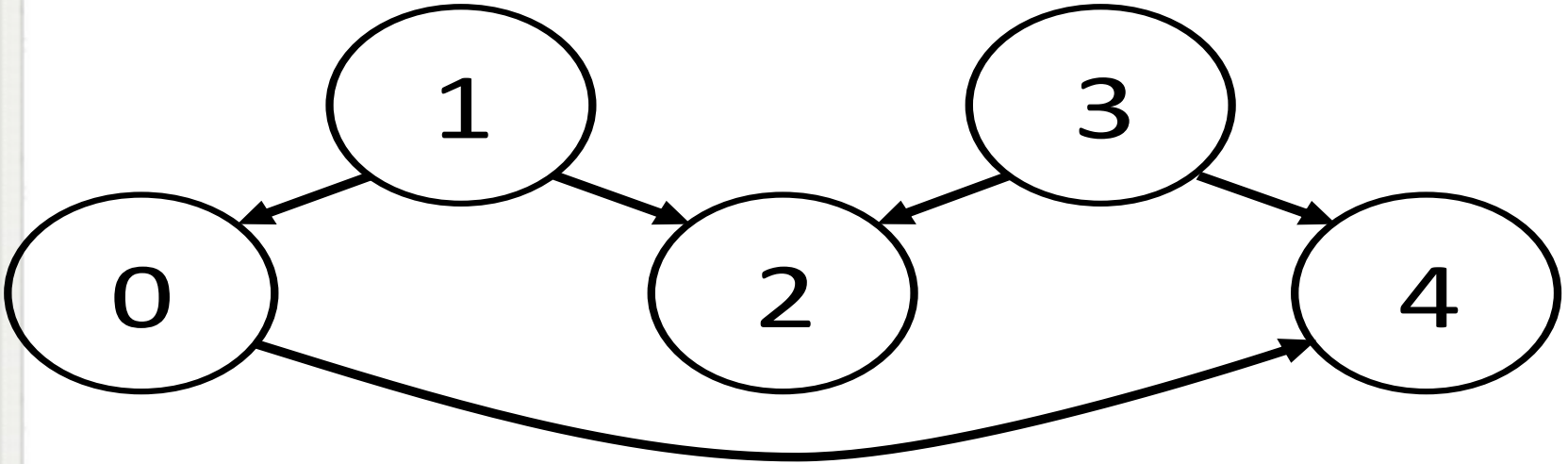
```
process Philosopher[4] {
    while (true) {
        /* desna pa leva */
        wait(fork[0]); wait[fork[4];
        eat;
        signal(fork[0]); signal(fork[4]);
        think;
    }
}
```



Graph-Based Protocols

- Нека је *R* скуп свих ресурса и нека су *ri* и *rj* два ресурса из датог скупа. Уколико се у дати скуп уведе релација парцијалног уређења (→) код које је ресурс *ri* пре ресурса *rj* (*ri*→*rj*) онда и сваки процес који приступа ресурсима *ri* и *rj* тим ресурсима мора да приступи тако што прво приступи ресурсу *ri* па онда ресурсу *rj*.
- Граф ресурса формиран на овај начин је директан ациклички граф.
- Граф ресурса би требало формирати на такав начин да води рачуна о томе који процес приступа којим ресурсима.

Dinning philosophers



Readers/Writers problem

- Процеси који читају (*Readers*) и процеси који пишу (*Writers*) деле базу података, датотеку, листу, табелу, ...
- *Readers* – само читају, *Writers* – читају и пишу
- Трансакције из конзистентног стања у конзистентно стање
- Писац захтева ексклузивни приступ, ако нема писаца, било који број читалаца
- Читаоци се такмиче са писцима, писци са другим писцима

Једноставан критичан регион

```
sem rw = 1;
process Reader[i = 1 to m] {
    while (true) { …
        wait(rw);
        read database;
        signal(rw);
    … }
}
process Writer[j = 1 to n] {
    while (true) { …
        wait(rw);
        write database;
        signal(rw);
    … }
}
```

Избегавање претераних ограничења

- Читаоци нису могли истовремено да читају по претходном решењу
- Идеја – само први читалац захтева међусобно искључивање
- Докле год има још читалаца који читају базу података – није потребно међусобно искључивање
- Потребан је бројач **активних** читаоца
- Инкрементирање и декрементирање бројача читаоца

Решење 1 *Readers/Writers – coarse grain*

```
int nr = 0; sem rw = 1;
process Reader[i = 1 to m] {
    while (true) { …
        < nr = nr + 1; if (nr == 1) wait(rw); >
        read database;
        < nr = nr - 1; if (nr == 0) signal(rw); >
    … }
}
process Writer[j = 1 to n] {
    while (true) { …
        wait(rw);
        write database;
        signal(rw);
    … }
}
```

Решење 1 – *fine grain*

```
int nr = 0; sem rw = 1; sem mutexR = 1;
process Reader[i = 1 to m] {
    while (true) { …
            wait(mutexR);
            nr = nr + 1;
            if (nr == 1) wait(rw);
            signal(mutexR)
            read database;
            wait(mutexR);
            nr = nr - 1;
            if (nr == 0) signal(rw);
            signal(mutexR);
    … }
}
```

Решење 1 – *fine grain*

```
process Writer[j = 1 to n] {
    while (true) { …
        wait(rw);
        write database;
        signal(rw);
    … }
}
```

Решење 1 - проблеми

- Предност се даје читаоцима, није равноправно
- Када једном почне низ читања, читаоци имају предност

- Друго решење које је засновано на расподељеним бинарним семафорима у којима се дефинише тачан предикат који укључује и број активних писаца
- RW: (nr == 0 or nw == 0) and nw <= 1

Coarse grain – predicate based

```
int nr = 0; nw = 0;
process Reader[i = 1 to m] {
    while (true) { …
        < await(nw == 0) nr = nr + 1; >
        read database;
        < nr = nr - 1; >
    … }
}
process Writer[j = 1 to n] {
    while (true) { …
        < await(nr == 0 and nw == 0) nw = nw + 1; >
        write database;
        < nw = nw - 1; >
    … }
}
```

Прослеђивање штафете

- Један семафор е за улазак у сваки атомски исказ
- Сваки boolean у <await(B) S;> је замењен са семафором и бројачем
- Семафори r и w и број закаснелих читалаца и писаца *dr* и *dw*
- Расподељени семафори – највише један може да буде 1 у сваком тренутку, *wait* је увек праћен са *signal*
- SIGNAL – кôд који дефинише за који семафор је *signal*

Signal кôд

```
if (nw == 0 and dr > 0) {
        dr = dr – 1; signal(r);
}
elseif (nr == 0 and nw == 0 and dw > 0) {
        dw = dw – 1; signal(w);
}
else
        signal (e);
```

Прослеђивање штафете – кôд (1)

```
int nr = 0, nw = 0; /* broj aktivnih readera i writera */
sem e = 1, r = 0, w = 0; /* raspodeljeni semafori */
int dr = 0; dw = 0; /* broj zakasnelih readera i writera */
process Reader[i = 1 to m] {
    while (true) { …
            wait(e);
            if (nw > 0) { dr = dr + 1; signal(e); wait(r); }
            nr = nr + 1;
            SIGNAL;
            read database;
            wait (e);
            nr = nr - 1;
            SIGNAL;
    … }
}
```

Прослеђивање штафете – кôд (1)

```
process Writer[j = 1 to n] {
    while (true) { …
            wait (e);
            if (nr > 0 or nw > 0) {dw = dw + 1; signal(e); wait(w);}
            nw = nw + 1;
            SIGNAL;
            write database;
            wait (e);
            nw = nw - 1;
            SIGNAL;
    … }
}
```

Поједностављење *Signal* кôда

- У процесу Reader – први сигнал nw == 0

  if (dr > 0) { dr = dr – 1; signal(r); }

  else signal(e);
- У процесу *Reader* – други сигнал nw == 0
- У процесу *Writer* – први сигнал nr == 0 i nw == 1
- У процесу *Writer* – други сигнал nw == 0 and nr == 0

Прослеђивање штафете комплетно

```
int nr = 0, nw = 0; /* Broj aktivnih readera i writera */
sem e = 1, r = 0, w = 0; /* Raspodeljeni semafori */
int dr = 0; dw = 0; /* Broj zakasnelih readera i writera */
```

Прослеђивање штафете комплетно

```
process Reader[i = 1 to m] {
    while (true) { …
            wait(e);
            if (nw > 0) { dr = dr + 1; signal(e); wait(r); }
            nr = nr + 1;
            if (dr > 0) {dr = dr - 1; signal(r); }
            else signal (e);
            read database;
            wait (e);
            nr = nr - 1;
            if (nr == 0 and dw > 0) { dw = dw – 1; signal(w);}
            else signal(e);
     … }
}
```

Прослеђивање штафете комплетно

```
process Writer[j = 1 to n] {
    while (true) { …
            wait (e);
            if (nr > 0 or nw > 0) {dw = dw + 1; signal(e); wait(w);}
            nw = nw + 1;
            signal (e);
            write database;
            wait (e);
            nw = nw - 1;
            if (dr > 0) { dr = dr – 1; signal(r); }
            elseif (dw > 0) { dw = dw – 1; signal(w);
            else signal (e);
    … }
}
```

Мењање преференци

- Читаоци се закашњавају ако писац чека
- Закаснели читалац се буди само ако нема писца који тренутно чека
- **Закашњавање читаоца** (*Reader*):

  if (nw > 0 or **dw > 0**)

  { dr = dr + 1; signal(e); wait(r); }
- **Буђење писца** (*Writer*):

  if (dw > 0) { dw = dw – 1; signal(w); }

  elseif (dr > 0) { dr = dr – 1; signal(r); }

  else signal(e);

Проблем алокације ресурса

- Када се процесу може дозволити приступ ресурсу?
- Приступ критичној секцији, штампачу, бази података, датотеци, локације у баферу ограничене величине, ...
- У критичној секцији – да ли процес има дозволу – не ком процесу се даје дозвола
- *Readers/Writers*, читаоци су имали предност, али не на нивоу појединог процеса
- Генерално решење где захтев има параметре – тип јединица и број јединица

Генерални захтев и ослобађање

```
request(parameters) :
        <await(request satisfied) take units;>
    ...
release(parameters) :
        <return units;>
```

- атомске акције су неопходне
- бројеви узетих и враћених јединица нису нужно исти.

Прослеђивање штафете

request(parameters) :
wait(e);
if (request not satisfied) *DELAY*;
take units;
SIGNAL;

release(parameters) :
wait(e);
return units;
SIGNAL;

DELAY и *SIGNAL* кôд

- **DELAY** изгледа као

if (request not satisfied)

{ dp = dp + 1; signal(e); wait(rs);}

где *је* ***rs request*** семафор

- за сваки ***delay condition*** (услов за закашњавање), постоји **посебан семафор**
- **SIGNAL** код зависи од типа проблема за алокацију ресурса

Shortest Job Next Allocation SJN

- један дељени ресурс
- **request(time,id), *time*** је ***integer*** – колико дуго
- **resurs free =>** одмах алокација
- **resurs not free => *delay queue*** поређан у складу са параметром ***time*** **–** временом колико ће процес да задржава ресурс
- ***release* –** ресурс се даје процесу у ***delay queue*** са минималном вредношћу параметра ***time***
- минимизација средњег времена завршетка посла

SJN предикат

- ***Unfair*** и старење процеса са великим временима – потребне модификације

- **free – *boolean*** и ***pairs* –** скуп записа **(*time,id*)** процеса поређаних по *time*

  SJN: **pairs** је уређен скуп ∧ **free => (pairs == 0)**

- ***Resurs*** не сме да буде ***free*** ако постоје захтеви који чекају

SJN coarse grain

- Ако игноришемо ***SJN*** политику

  bool free = true;

  request(time,id) : <await(free) free = false;>

  release() : <free = true;>
- Потребно је за процес са најмањим ***time***
- ***Fine grain*** решење засновано на расподељеним семафорима

Fine grain решење - базично

request(time,id) :
wait(e);
if (!free) *DELAY*;
free = false;
SIGNAL;
release() :
wait(e);
free = true;
SIGNAL;

SJN кроз *DELAY* и *SIGNAL*

- **DELAY:** уметни уређени пар параметара ***time*** и ***id*** у скуп *pairs* закашњених процеса и ослободи критичну секцију са ***signal(e)***, закасни на семафору за ***request***
- Сваки процес има различит ***delay*** услов зависно од положаја у ***pairs*** скупу
- **b[n]** низ семафора за **n** процеса – ***id*** у опсегу 0 до **n-1**
- **DELAY** укључује уређивање ***pairs*** скупа

Приватан семафор (1)

```
bool free = true;
sem e = 1, b[n] = ([n] 0);
typedef Pairs = set of (int, int);
Pairs pairs = ∅;
request(time,id) :
wait(e);
if (!free) {
    umetni (time,id) u pairs;
    signal(e); wait(b[id]);}
free = false;
signal(e);
```

Приватан семафор (2)

release() :

wait(e);

free = **true**;

if (pairs != ∅) {

ukloni prvi uređeni par (time,id) iz skupa pairs;

signal(b[id]); /* prosleđivanje štafete do procesa sa određenim proces id */}

else signal(e);

Генерализација алокације коришћењем прослеђивања штафете

- Замена *boolean free* са *integer* ***available***
- *Request* – испитивање ***amount <= available*** и алокација неког броја *units*
- *Release* – повећавање ***available*** са ***amount*, и испитивање *amount <= available* за** процес који чека са најмањом вредношћу time. Ако *true* => алокација, у супротном *signal(e)*

Баријере са семафорима

- *Signaling semaphore* – иницијално 0, један процес извршава signal(s) да синхронизује други, а други чека са wait(s).
- Баријера за два процеса:

```
sem arrive1 = 0; sem arrive2 = 0;
process Worker1 {…
    signal(arrive1);
    wait(arrive2);
…}
process Worker2 {…
    signal(arrive2);
    wait(arrive1);
…}
```

Проблеми са семафорима

- *wait* и *signal* операције треба да буду у паровима, раширено по коду
- Извршавање операција на погрешном семафору може да изазове *deadlock*, није заштићен критичном секцијом, …
- За међусобно искључивање и условну синхронизацију се користе исте примитиве.

Регионы

Основни појмови

- Код програмске парадигме критичних региона полази се од чињенице да се у конкурентним програмима често јавља потреба за приступ критичној секцији. Основна одлика ове програмске парадигме је увођење посебне синтаксе за експлицитно означавање критичних секција, који се у овом контексту називају – критични региони. Код приступа критичном региону је имплицитно обезбеђено међусобно искључивање процеса.

Основни појмови

- Критични регион се синтаксно дефинише на следећи начин:

```
region res do
begin
    ...
end;
```

Јава

```
synchronized (referenca){
        ...
}
```

Основни појмови

- *Условни критични регион* је критични регион који поред обезбеђивања међусобног искључивања има и механизам за условну синхронизацију процеса преко (опционих) *await* наредби.
- Када се унутар критичног региона наиђе на *await* наредбу чији услов није задовољен, процес се блокира и притом одриче ексклузивног права приступа ресурсу, јер неки други процес можда чека да уђе у тај критични регион да би омогућио да се услов испуни (тиме се избегава могућа ситуација да се два процеса међусобно бескрајно дуго чекају и настајање мртвог блокирања (*deadlock*)).

Основни појмови

- Након испуњења датог услова и поновног добијања права ексклузивног приступа, процес се деблокира и наставља рад. По изласку из критичног региона, један од процеса који су били блокирани чекајући да уђу у критични регион се деблокира и добија право ексклузивног приступа региону. Тај поступак се понавља док год има процеса који чекају на улазак у критични регион.

Основни појмови

- Условни критични регион се синтаксно дефинише на следећи начин:

```
region res do
begin
    ...
    await(condition);
 ...
end;
```

Јава – могућа имплементација

```
synchronized (referenca){
        ...
        referenca.notifyAll();
        while(!condition)
                reference.wait();
        ...
        referenca.notifyAll();
}
```

Монитори

Монитори

- Механизам апстракције података – енкапсулација репрезентације апстрактних објеката – скуп операција је једини пут приступа објектима
- Монитори – апстракција података + процеси који позивају мониторске процедуре
- Међусобно искључивање – имплицитно – као <*await*>
- Синхронизација - *condition* променљиве

Модуларизација

- Активни процеси и пасивни монитор
- Све што се ради у активним процесима су позиви процедура монитора – сва конкурентност је сакривена у монитору за програмере активних процеса
- Програмери монитора могу да мењају имплементацију, докле год се ефекти не мењају
- Лако разграничавање одговорности међу програмерима

Структура монитора

```
monitor mname {
    декларација сталних променљивих
    иницијализација
    процедуре
}
```

- Сталне променљиве – задржавају своје вредности
- Само имена процедура су видљива преко позива mname.opname(argumenti)

Енкапсулација

- Приступ само кроз позиве процедура
- Правила видљивости унутар монитора – нема референцирања изван монитора
- Сталне променљиве су иницијализоване пре позива процедура
- Програмер монитора не може да зна редослед позивања мониторских процедура
- Мониторска инваријанта – предикат смислених стања, успостављених иницијализацијом, одржаваних процедурама

Међусобно искључивање

- Мониторска процедура је активна ако процес извршава исказ у процедури
- Највише једна инстанца мониторске процедуре је активна у једном тренутку
- У пракси, најчешће скривени семафори обезбеђују међусобно искључивање

Условне променљиве

- Закашњавање ако стање монитора не задовољава неки *boolean* услов
- cond cv;
- cv је queue закашњених процеса који нису директно видљиви програмеру

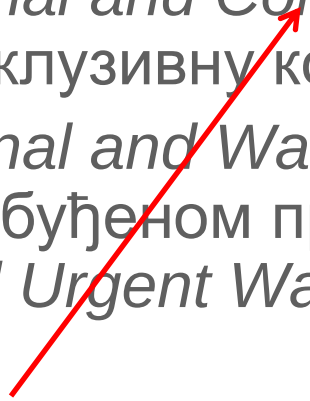
- empty(cv) функција за испитивање стања реда чекања

- wait(cv) кашњење и одрицање ексклузивног приступа – други процеси улазе

- signal(cv) сигнал буди процес који је на почетку реда чекања – у супротном **нема ефекта**

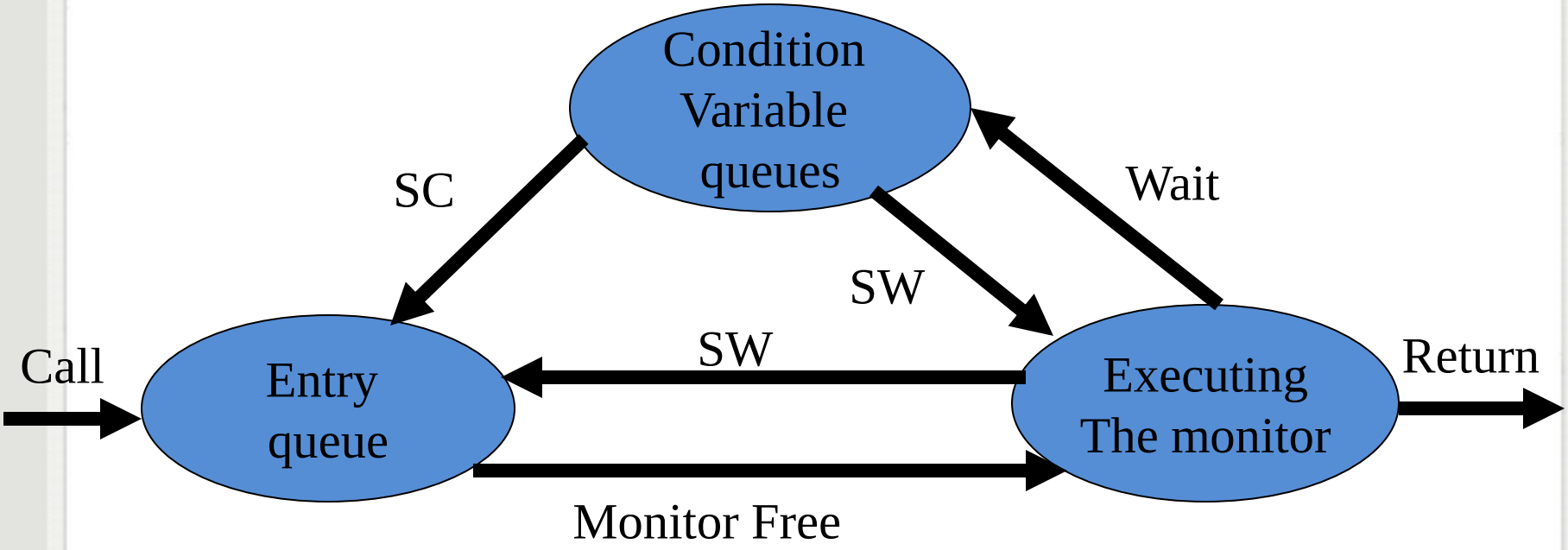
Сигнал дисциплине

- Процес долази до сигнал и има имплицитно закључавање (међусобно искључивање) – шта је следеће ако сигнал буди други процес
- *Signal and Continue* – *nonpreemptive*, задржава ексклузивну контролу
- *Signal and Wait* – *preemptive*, прослеђује контролу пробуђеном процесу, одлази на ред чекања (*Signal and Urgent Wait* – Приоритет)



Јава

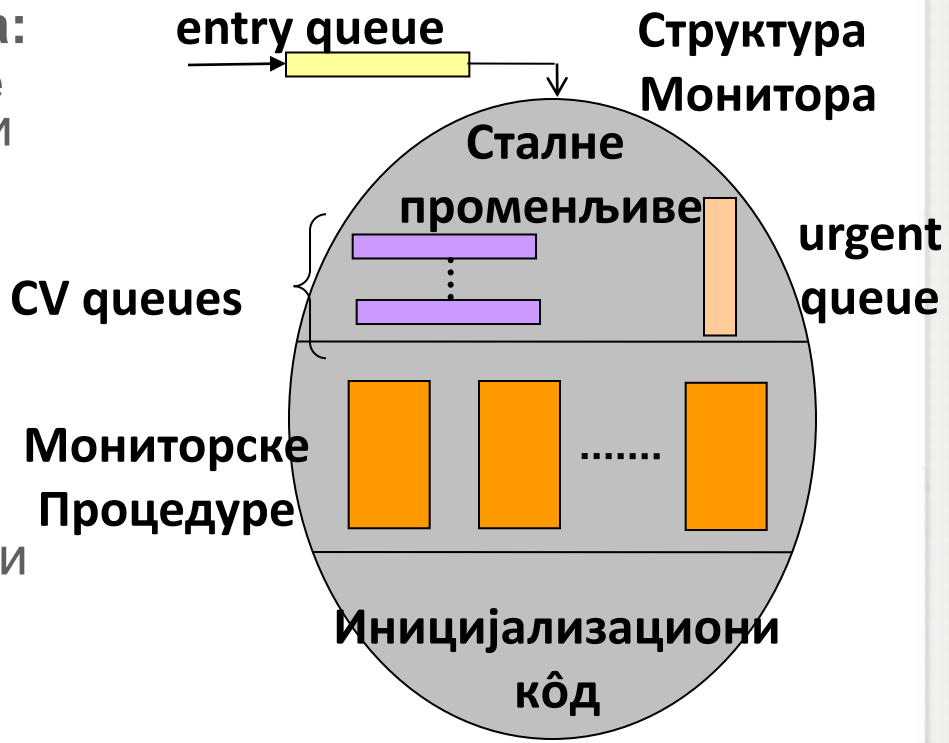
Дијаграм стања за мониторе



Редови у мониторима

Монитори одржавају пуно различитих редова чекања:

- *Entry Queue* → Постоји да се поређају процеси и обезбеди међусобно искључивање у приступу монитору.
- *CV Queues* → редови за суспендоване процесе који чекају на испуњење услова.
- *Urgent Queue* → Да се имплементира *Signal and Urgent Wait* – ређа процесе који су дали сигнал другима и често треба да напусте монитор.



Имплементација SW монитора помоћу семафора

Монитор помоћу семафора:

```
wait(mutex)
//највише један процес активан у
    монитору, међусобно
    искључивање мониторских
    procedura
…
  тела мониторских процедура
…
if next.count > 0 signal(next);
else signal(mutex);
// next је семафор који служи за
    urgent queue !!, а next.count
    бројач закаснелих на urgent
    queue
```

CV.wait: →

```
 CV.count ++;
 if next.count > 0 signal(next);
 else signal (mutex);
 wait(CV_sem);
 CV.count --;
```

CV.signal: →

```
 if CV.count > 0 {
    next.count ++;
    signal(CV_sem);
    wait(next);
    next.count --;
 }
```

Монитор за семафоре

```
monitor Semaphore {
    int s = 0; /* ili neka druga inicijalna vrednost */
    cond pos; /* signal kada je s inkrementirano */
    procedure semWait() {
        while (s == 0) wait(pos);
        s := s-1;
    }
    procedure semSignal() {
        s:= s + 1;
        signal(pos);
    }
}
```

Signal and Continue за семафоре

- Предикат s >= 0 мора да буде очуван
- После *signal*, процес који извршава *semSignal* наставља – вредност s није неминовно > 0 када пробуђени процес добије *lock*
- То је разлог због кога је *while* петља неопходна – да се поново испита s!
- Није гарантовано FIFO семафор – више процеса може да буде у *entry queue* – после чекања на условној променљивој *queue*

Signal and Wait за семафоре

- FIFO је загарантован – директно на основу условне променљиве *queue*
- Нема потребе за *while* петљом – поновно испитивање вредности s није неопходно =>

  if (s == 0) wait(pos);
- Процес из *pos queue*, који је добио *Signal*, извршава се одмах
- Такође делови процедура између *signal* операција су критичне секције

Проверити да ли је у конкретној имплементацији!!!

Прослеђивање услова - *FIFO* семафор

```
monitor FIFOsemaphore {
    int s = 0; /* ili druga inicijalna vrednost */
    cond pos; /* signal kada bi trebalo s > 0 */
    procedure semWait() {
        if (s == 0)         wait(pos);
        else                s := s-1;
    }

    procedure semSignal() {
        if (empty(pos))s:= s + 1;
        else                signal(pos);
    }
}
```

Особине прослеђивања услова-*Passing the condition*

- Сада се не мења стална променљива s када има закаснелих, већ прослеђивање услова види **само** процес који је дошао на ред у ***CV queue***
- Нема могућности други процес да види промену – само монитор!
- Нема *while* петље ни за *SC*
- Гарантован *FIFO*, ако се користи *FIFO CV*

Signal and Continue

- Користи се у *Unix* и Јави
- Компатибилно са распоређивањем на основу приоритета
- Једноставнија семантика – комплетне процедуре су критичне секције
- *wait(cv, rank)* – најмањи *rank* - приоритет
- *signal_all(cv)* – пробуди све процесе
- *minrank(cv)* – *rank* на почетку queue

Bounded buffer (1)

```
monitor Bounded_Buffer {
    typeT buf[n];
    int front = 0, rear = 0, count = 0;
    cond not_full, not_empty;

    procedure deposit(typeT data) {
        while (count == n) wait(not_full);
        buf[rear] = data; rear = (rear + 1) % n;
        count++;
        signal(not_empty);
    }
```

Bounded buffer (2)

```
    procedure fetch(typeT &result) {
        while (count == 0) wait(not_empty);
        result = buf[front]; front = (front + 1) % n;
        count--;
        signal(not_full);
    }
}
```

Readers/Writers – broadcast signal

- Монитор само даје дозволе за приступ
- *Request* и *release* за *read* и *write*
- Број читалаца и писаца је *nr* и *nw*
- Предикат – мониторска инваријанта

$(nr == 0 \vee nw == 0) \wedge nw <= 1$

- Код технике *broadcast signal* се буде све заинтересоване групе блокираних процеса. Не одлучује се која ће бити наредна група. То се препушта распоређивачу у редовима.

Readers/Writers monitor (1)

```
monitor RW_Controller {
    int nr = 0, nw = 0;
    cond oktoread; /* signaled when nw == 0 */
    cond oktowrite; /* both nr and nw are 0 */

    procedure request_read() {
        while (nw > 0) wait(oktoread);
        nr = nr + 1;
    }
```

Readers/Writers monitor (2)

```
    procedure release_read() {
            nr = nr - 1;
            if (nr == 0) signal(oktowrite);
    }
    procedure request_write() {
            while (nr > 0 || nw > 0) wait(oktowrite);
            nw = nw + 1;
    }
    procedure release_write() {
            nw = nw - 1;
            signal(oktowrite);
            signal_all (oktoread);
    }
}
```

Обавештава све блокиране групе (*broadcast signal*), не одређује која ће бити наредна.

Shortest next allocation monitor

```
monitor Shortest_Job_Next {
    bool free = true;
    cond turn;
    procedure request(int time) {
        if (free) free = false;
        else wait(turn, time);
    }
    procedure release()
        if (empty(turn)) free = true
        else signal(turn);
    }
}
```

Passing the condition

Interval Timer Monitor

- Две операције (процедуре)

delay (interval) и *tick* – процес кога буди хардверски тајмер са високим приоритетом

wake_time = tod + interval;

- Сваки процес који позива – приватно време буђења *wake_time*
- По једна условна променљива за сваки процес?
- Када је *tick*, стална променљива се проверава и ради се сигнал ако је услов испуњен – гломазно

Прекривање услова – *Covering condition*

- Једна условна променљива
- Када било који од прекривених услова (*covered condition*) може да постане истинит – сви процеси се буде
- Сваки процес поново испитује своју условну променљиву
- Буђење и процеса који могу да пронађу да је *delay condition false*

Covering condition monitor

```
monitor Timer {
    int tod = 0;  cond check;
    procedure delay(int interval) {
        int wake_time;
        wake_time = tod + interval;
        while (wake_time > tod) wait(check);
    }
    procedure tick() {
        tod = tod + 1;
        signal_all(check);
    }
}
```

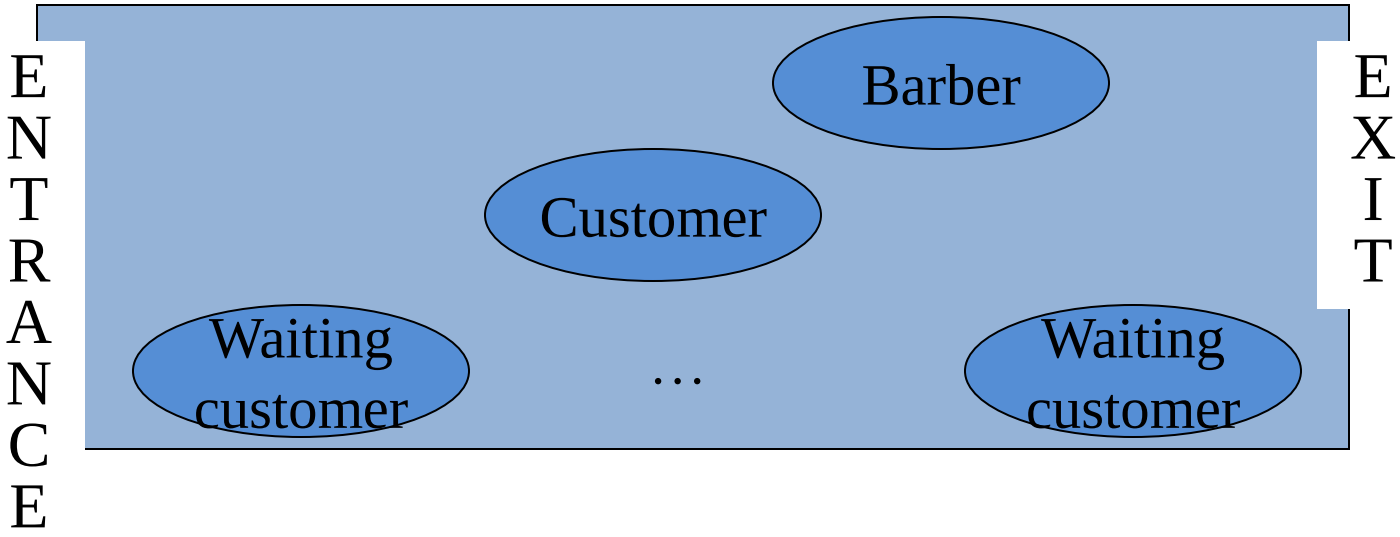
Priority Wait Timer

```
monitor Timer {
    int tod = 0;  cond check;
    procedure delay(int interval) {
        int wake_time;
        wake_time = tod + interval;
        if (wake_time > tod) wait(check, wake_time);
    }
    procedure tick() {
        tod = tod + 1;
        while (!empty(check) && minrank(check) <= tod)
        signal(check);
    }
}
```

Предности *priority wait timer*

- Процеси су поређани у складу са временима буђења
- *minrank* се користи за одређивање да ли да се пробуди следећи процес који је закашњен
- *while* петља за *signal* је неопходна за случај када неколико процеса има исто *wake_time*
- Компактно, ефикасно – засновано на статичком редоследу између *delay* услова

Sleeping Barber Rendezvous



Client/Server relationship

- Фризерска радња као монитор, клијенти захтевају шишање – процеси, брица – процес
- Процедуре *get_haircut*, *get_next_customer*, *finished_cut*
- Брица – *get_next_customer* и *finished_cut* за напуштање радње
- Потреба за синхронизацију брице и клијента – *rendezvous* – баријера за два процеса (али брица са било којим клијентом)

Синхронизациона стања

- Клијенти – седење у столици и напуштање
- Брица – постаје расположив, шишање и напуштање
- Праћење стања коришћењем бројача, иницијално 0 и инкремент за бележење колико је процеса достигло то стање:
  - cinchair – клијената село у столицу
  - cleave – клијената напустло
  - bavail – колико је пута берберин био расположив
  - bbusy – колико је пута берберин шишао
  - bdone – колико је пута берберин испратио некога

Синхронизациона стања

- Релације важе између услова: cinchair и cleave, bavail, bbusy и bdone

C1: cinchair >= cleave $\wedge$ bavail >= bbusy >= bdone

C2: cinchair <= bavail $\wedge$ bbusy <= cinchair

C3: cleave <= bdone

C1 $\wedge$ C2 $\wedge$ C3

Редефинисање бројача

- Проблем бесконачног инкрементирања
- Промена променљивих увођењем разлика

barber == bavail – cinchair

chair == cinchar – bbusy

open == bdone – cleave

Вредности су само 0 и 1

Монитор *Barber_Shop*

```
int barber = 0, chair = 0, open = 0;
cond barber_available; /* signal kada barber > 0 */
cond chair_occupied; /* signal kada chair > 0 */
cond door_open; /* signal kada open > 0 */
cond customer_left; /* signal kada open == 0 */
/* četiri sinhronizaciona uslova: za customera - barber
    raspoloživ i barber da otvori vrata, za barbera –
    customer da stiže i customer da napušta */
```

Монитор *Barber_Shop* (1)

```
procedure get_haircut() {
        while (barber == 0) wait(barber_available);
        barber = barber – 1;
        chair = chair + 1;
        signal(chair_occupied);
        while (open == 0) wait(door_open);
        open = open – 1;
        signal(customer_left);
}
```

Монитор *Barber_Shop* (2)

```
    procedure get_next_customer() {
            barber = barber +1;
            signal(barber_available);
            while (chair == 0) wait(chair_occupied);
            chair = chair – 1;
    }
    procedure finished_cut() {
            open = open + 1;
            signal(door_open);
            while (open > 0) wait(customer_left);
    }
}
```

Проблем кружног тока

- Раскрсница кружног тока има 3 двосмерне улице са по једном саобраћајном траком у сваком смеру повезане на кружни ток. Предност у кружном току имају возила која се већ налазе у кружном току. Реализујте монитор у коме ће се регулисати саобраћај кружног тока. Сматрајте да се, ако у претходном сегменту кружног тока има возила, мора чекати на улазу у кружни ток. Такође, возила морају чекати возила испред себе на улазној улици у кружни ток (једна улазна саобраћајна трака).

Проблем кружног тока

- У решењу, непотребно задржавање процеса за возила у монитору није дозвољено. Позивање више различитих мониторских процедура за процесе возила је дозвољено. Смер кретања у кружном току је прва_ул-друга_ул-трећа_ул-прва_ул. Претпоставите да сегмент кружног тока може да прими неограничен број возила.

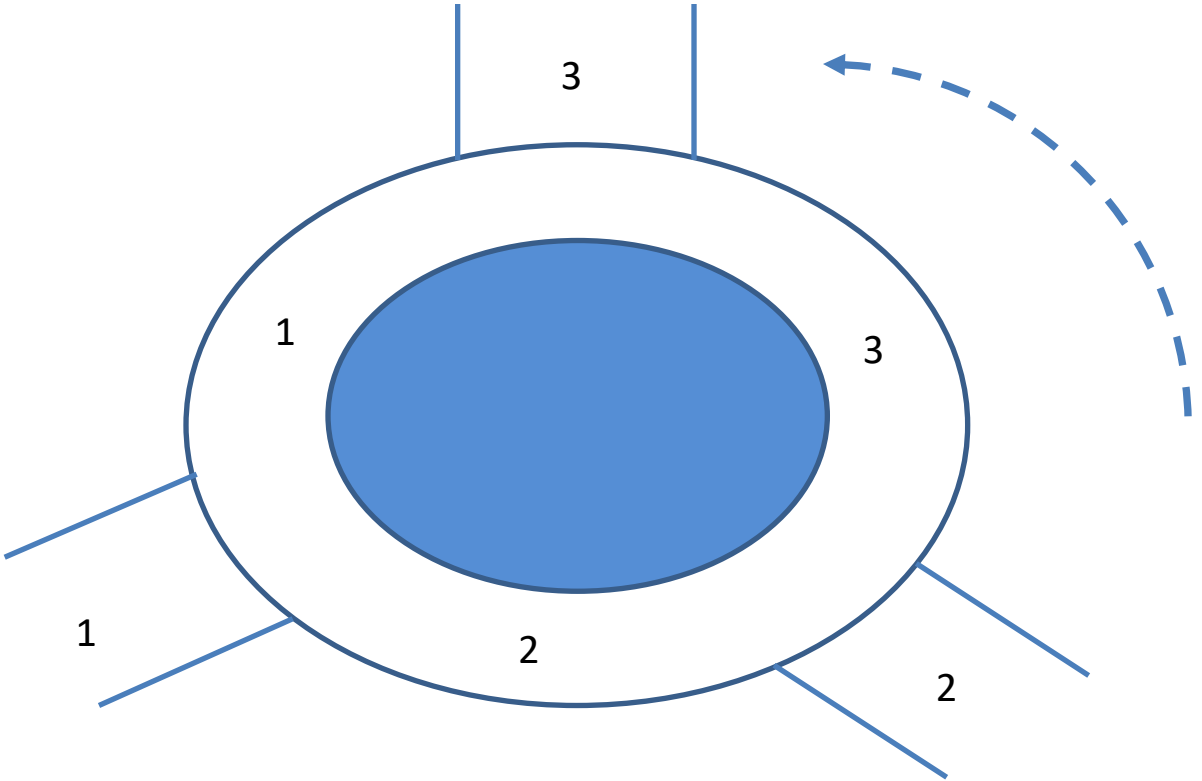
Почетни коментари

- У поставци су реченице: У решењу, непотребно задржавање процеса за возила у монитору није дозвољено. Позивање више различитих мониторских процедура за процесе возила је дозвољено. – **ВИШАК**
- Када би се правиле процедуре типа: **од-улаза-до-излаза**, процес би непотребно задржавао монитор и **дозвољавала би да само једно возило буде у раскрсници** – задржавање скривене критичне секције јако дуго. Ни сам процес за возило не би могао да ради.
- Када би се правиле процедуре од **улаза сегмента до излаза тог сегмента**, **задржавао би се монитор све време док возило путује кроз сегмент** – многи процеси би чекали да започну мониторску процедуру, па би чекања када има више возила била примарно због задржавања у ентру qyeye монитора

Једино решење за процедуре

- Процедуре за улаз у сегмент и напуштање сегмента сигурно ако се пролази само један сегмент
- Шта ако се пролази кроз више сегмената?
- Прелаз из сегмента у сегмент је тренутан – треба увести додатну процедуру за прелазак из сегмента у сегмент **јер нема задржавања монитора**!!!

Нумерисање сегмената и улаза



Позивајући процеси

- Шта са позивајућим процесима?
- Ако је један сегмент (улази на 1 а излази на 2):
  call kruzni_tok.start (1),
  radi nešto malo lokalno,
  call kruzni_tok.leave (2)
- За два сегмента (улази на 1 а излази на 3):
  call kruzni_tok.start (1),
  radi nešto malo lokalno,
  call kruzni_tok.moveFrom(2)
  radi nešto malo lokalno и
  call kruzni_tok.leave (3)

Како одржати *FIFO queue* на улазу

- *FIFO* на улазу у кружни ток је обезбеђен у *Signal and Wait* дисциплини
- Решење које обезбеђује *FIFO* за *Signal and Continue* је комплексније
- Ако је могућ избор дисциплине – *Signal and Wait*

Монитор *kruzni_tok – Signal and Wait*

```
monitor kruzni_tok{
const N = 3;
int[1..N] count;
condition [1..N] enter;

procedure start(int segment){
        if ((enter[segment].queue()) or (count[segment] <> 0))
                enter[segment].wait(); //sta ako condition nije FIFO?
        count[(segment mod N)+1]= count[(segment mod N)+1]+1;
}
```

Монитор *kruzni_tok – Signal and Wait*

```
procedure leave(int segment){
        count[segment]=count[segment]-1;
        while((enter[segment].queue())
                and (count[segment] = 0))
                        enter[segment].signal();
}
```

Монитор *kruzni_tok – Signal and Wait*

```
procedure moveFrom(int segment){
        count[(segment mod N)+1] =count[(segment mod N)+1]+1;
        count[segment] = count[segment] - 1;
        while((enter[segment].queue())
                and (count[segment] = 0))
                        enter[segment].signal();
}
иницијализација
{
        for(i = 1; i < N; i++) count[i] = 0;
}
}
```

Питања?