Синхронизация на процеси

проф. д-р инж. Христо Вълчанов

http://cs.tu-varna.bg

Достъп до общи ресурси

- Едновременния достъп до общи ресурси може да доведе до некоректни резултати.
- Необходими са механизми за осигуряване правилният ред на достъп до общите ресурси.
- Общи ресурси – памет, устройства.

Критична секция

- **Критична секция (Critical Section - CS)** – тази част от кода на процес, в който има обръщение към общ ресурс.

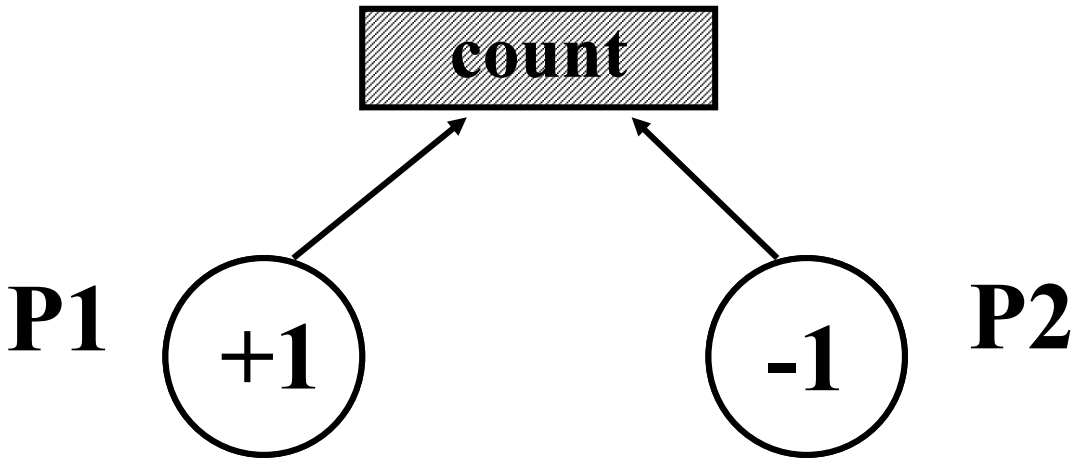
- Изисквания към CS:
  - В критична секция в даден момент може да се намира само един процес.
  - Ако един или няколко процеса желаят да влязат в критичните си секции, то на един от тях това задължително трябва да бъде разрешено.
  - Всеки процес може да преминава в произволно състояние извън критичната си секция, т.е. може да бъде дори спрян. Това не трябва да оказва влияние върху другите процеси при използване на общия ресурс.
  - Относителните скорости на развитие на процесите са предварително неизвестни и произволни.
  - Процес не може да спре изпълнението си в критична секция.
  - Критичните секции не трябва да имат приоритети.

Обща структура на процес

```
Process:
        while (true) {

            non_critical_section

            entry_section

            critical_section

            exit_section

            non_critical _section

        }
```

Пример за достъп до общ ресурс

Процес P1 извършва увеличава променливата **`count`** 1000000 пъти, процес P2 намалява променливата **`count`** 1000000 пъти. Началната стойност на **`count`** е 0.



- **Каква е стойността на *`count`* след завършване на процесите?**

Пример за достъп до общ ресурс (2)

```
shared int count;
```

```
/* P1.c */

int main () {

   for (i=0; i<1000000; i++) {
        . . .
        count++;
        . . .
   }
}
```

```
/* P2.c */

int main () {

   for (i=0; i<1000000; i++) {
        . . .
        count--;
        . . .
   }
}
```

Пример за достъп до общ ресурс (3)

- След компилация:

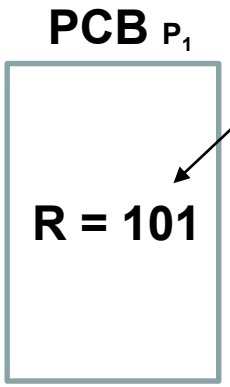
```
P1:     LOADR   count       Зарежда count в регистър R
        INCR                Инкрементира R
        STORER  count       Записва R на адрес count

P2:     LOADR   count       Зарежда count в регистър R
        DECR                Декрементира R
        STORER  count       Записва R на адрес count
```

Влияние на прекъсванията

quant	P1	count	R	P2
t_1	. . .	100	???	. . .
t_2	LOADR count INCR	100 100	**100** **101**	

Влияние на прекъсванията (2)

quant	P1	count	R	P2
t_1	. . .	100	???	. . .
t_2	LOADR count INCR	100 100	100 101	



Влияние на прекъсванията (3)

quant	P1	count	R	P2
t_1	. . .	100	???	. . .
t_2	LOADR count INCR	100 100	100 101	
t_3		100 100 **99**	**100** **99** 99	LOADR count DECR STORER count

PCB P_1

R = 101

Влияние на прекъсванията (4)

quant	P1	count	R	P2
t_1	. . .	100	???	. . .
t_2	LOADR count INCR	100 100	100 101	
t_3		100 100 **99**	**100** **99** 99	LOADR count DECR STORER count

прекъсване

PCB P_1

R = 101

PCB P_2

R = 99

Влияние на прекъсванията (5)

quant	P1	count	R	P2
t_1	. . .	100	???	. . .
t_2	LOADR count INCR	100 100	100 101	
t_3		100 100 99	100 99 99	LOADR count DECR STORER count
t_4	STORER count	99 **101**	**101** 101	

PCB P_1

R = 101

PCB P_2

R = 99

Програмна защита на CS

Използване на променлива за ред на достъп

```
int turn=1;

P1:     for (;;) {
          while (turn==2) ;
          CS1;
          turn=2;
          Prog1;
        }
P2:     for (;;) {
          while (turn==1) ;
          CS2;
          turn=1;
          Prog2;
        }
```

Ако един от процесите се изпълнява безкрайно извън своята CS или спре, другият не може да влезе в своята.

Програмна защита на CS - 2

Използване на 2 променливи за ред на достъп

```
int k1=k2=false;

P1:     for (;;) {
          while (K2) ;
          K1=true;
          CS1;
          K1=false;
          Prog1;
        }
P2:     for (;;) {
          while (K1) ;
          K2=true;
          CS2;
          K2=false;
          Prog2;
        }
```

Ако са с еднакви скорости и двата едновременно могат да влязат в своите CS.

Програмна защита на CS - 3

Използване на 2 променливи за ред на достъп и заявка

```
int k1=k2=false;

P1:     for (;;) {
          K1=true;      // заявка за влизане
          while (K2) ;
          CS1;
          K1=false;
          Prog1;
        }
P2:     for (;;) {
          K2=true;
          while (K1) ;
          CS2;
          K2=false;
          Prog2;
        }
```

Ако са с еднакви скорости и двата могат безкрайно да чакат на входа на своите CS.

Алгоритъм на Декер

Използване на 2 флага за заявка и един за ред на достъп

```
C1=true;
C2=true;
queue=1;

// Process 1
...
C1=false;
while !C2 {
        if (queue == 2) {
                C1=true;
                while (queue == 2) ;
                C1:=false;
        }
}
CS1;
queue=2;
C1=true;
...
```

Прекалено сложен за ефективно използване, особено при повече от 2 процеса.

Алгоритъм на Петерсон

Използване на 2 флага за заявка и един за ред на достъп

```
C1=true;
C2=true;
queue=1;

// Process 1
...
C1 = false;
queue = 2;
while (!C2) && (queue == 2) do ;
CS1;
C1 = true;
...
```

Сложен за използване от много процеси.

Синхронизация чрез забраняване на прекъсвания

- Забраняване на прекъсванията докато процес е в критична секция.

```
while (true) {
        disable_interrupts
        critical section
        enable_interrupts
        remainder section
};
```

Синхронизация чрез заключвания

Решенията (хардуерни) се базират на механизма на заключванията (***locks***).

```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (TRUE);
```

Синхронизационен хардуер

- Наличие на **атомарни** хардуерни инструкции за реализиране на заключванията.
- **Атомарен = непрекъсваем.**
- Инструкция за тестване на съдържанието на памет и установяване на стойност.

Инструкция test_and_set

Дефиниция:

```
boolean test_and_set (boolean *target) {
                boolean rv = *target;
                *target = TRUE;
                return rv:
}
```

1. Изпълнява се атомарно;
2. Връща стойността на изпратената като параметър променлива
3. Установява стойност *true* на променливата

Синхронизация с test_and_set

```
do {
      while (test_and_set(&lock)) ;
      CS;
      lock = false;
      Prog;
} while (true);
```

- Обща променлива *lock*;
- Всеки процес изпълнява този код
- Активно изчакване (заема се CPU)

Mutex locks

- Предишните решения са сложни и неудобни за използване
- Разработени са софтуерни средства за решаването на проблема
- Просто средство е т.н. *mutex lock*.
- Mutex lock притежава булева променлива, асоциирана с него. Указва дали е налично заключване.

```
acquire() {
  while (!available) ; /* busy wait */
  available = false;
}

release() {
     available = true;
}
```

Mutex locks

```
acquire() {
       while (!available) ; /* busy wait */
        available = false;
}
release() {
        available = true;
}

do {
    acquire lock
      CS
    release lock
      Prog
 } while (true) ;
```

- Реализират се обикновено чрез хардуерни инструкции
- Активно изчакване (заема се CPU) - **spinlock**

Семафори

- **Семафор** – специален вид целочислена положителна променлива, върху която са определени две операции (примитиви) P и V.
- Операциите са неделими.
- Семафор, приемащ само две стойности 0 и 1 – **двоичен семафор.**

Семафори (2)

- **операция P(S):**

Процесът се опитва да изпълни S=S-1. При това са възможни два случая:

  - S>0. Процесът изпълнява S=S-1 и продължава своето развитие;
  - S=0. Процесът се блокира (wait) докато S стане по-голямо от нула. Тогава и само тогава той завършва операцията S=S-1 и продължава изпълнението си.

- **операция V(S):**

Процесът изпълнява операцията S=S+1. При това също са възможни два случая:

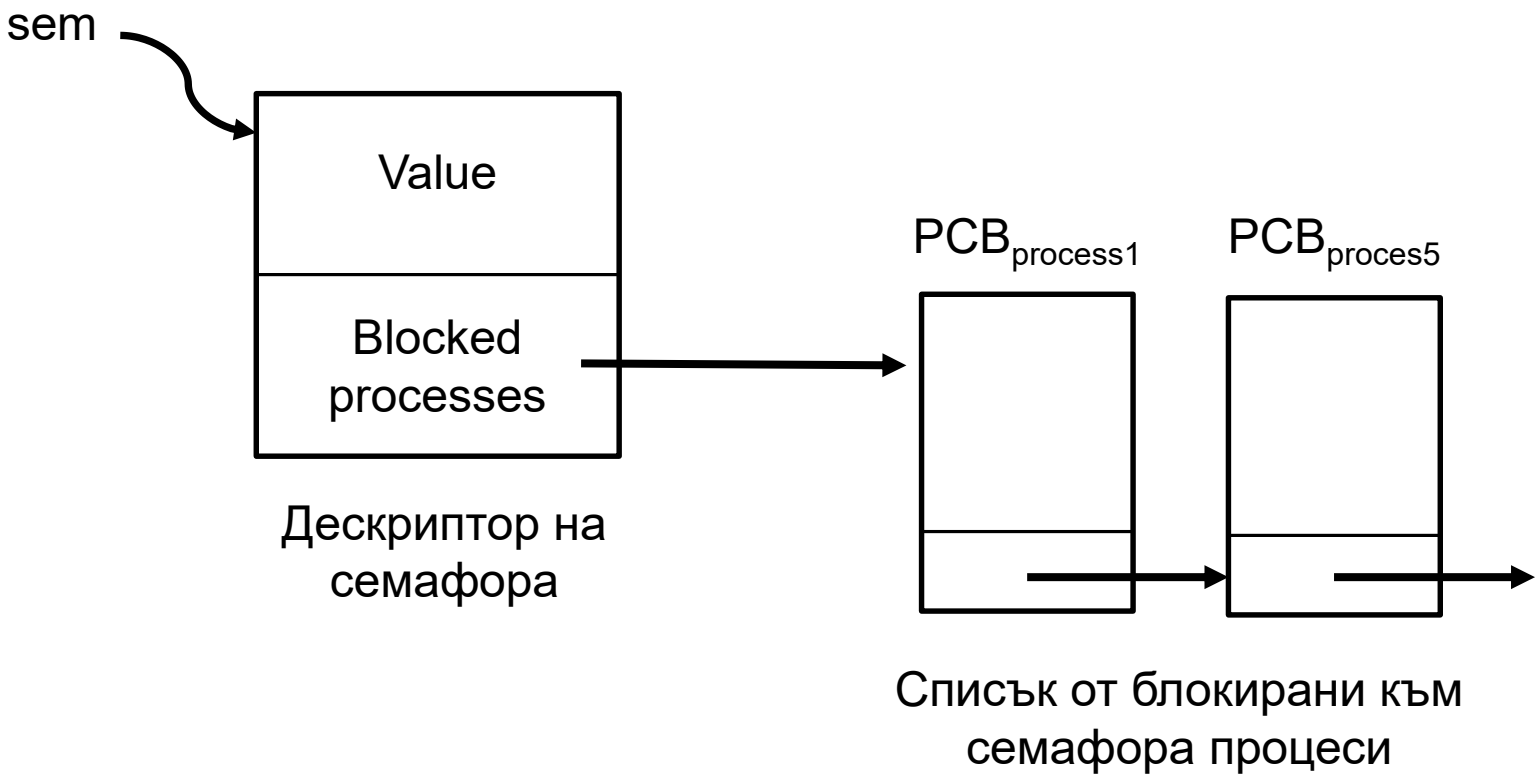
  - S>0. Процесът, изпълняващ операцията V(S), продължава да работи;
  - S=0. Процесът, изпълняващ операцията V(S), продължава да работи, но преди това се активира един от блокираните процеси (т.е. дава му се възможност да завърши започнатата си P(S) операция).

Използване на семафори

```
semaphore sem=1;


P1:     for (;;) {
          P(sem);
          CS1;
          V(sem);
          Prog1;
        }
P2:     for (;;) {
          P(sem);
          CS2;
          V(sem);
          Prog2;
        }
```

Семафор с повече от две стойности – общ семафор

Реализация на семафор

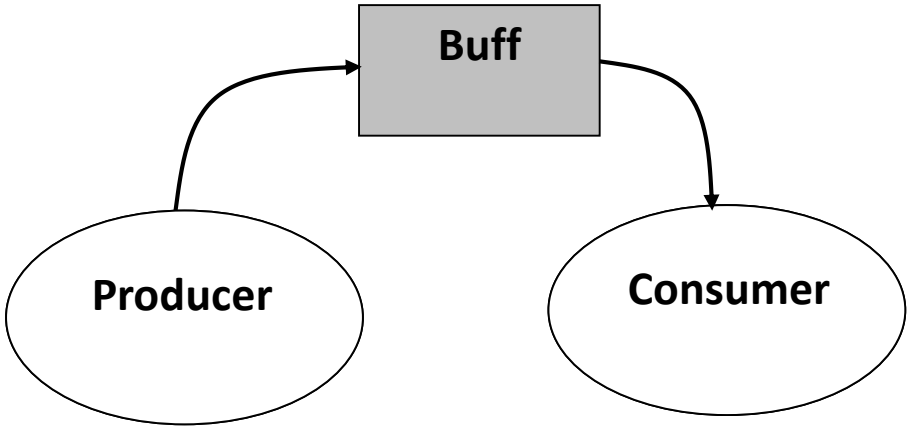


Реализация на семафорни операции

```
void P(semaphore S) {
   if (S->Value == 0) {
        извикване_диспечера_за_блокиране_текущ_процес;
        (S->Value)--;
   } else {
        (S->Value)--;
   }
}

void V(semaphore S) {
   (S->Value)++;
   if (S->BlockedProcesses != NULL) {
        преместване_първия_процес_в_ReadyQueue;
   }
}
```

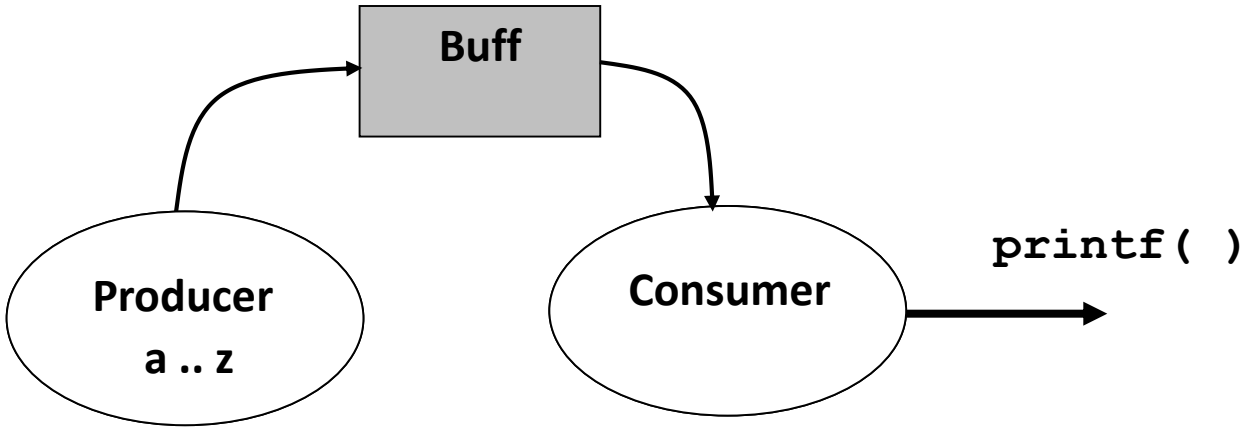
Задача “Производител-консуматор”

Производителят генерира данни и ги записва в буфер с ограничен размер.
Консуматорът чете данните от буфера и ги обработва.



Пример “Производител-консуматор”

Производителят генерира азбуката и я записва символ по символ в буфер от тип *char*.
Консуматорът чете данните от буфера и ги извежда на екрана.



Пример “Производител-консуматор”

Възможни варианти на извеждането:

- a a a a a a a b b c d d d d d d d e e …
- b f p z
- z

➢ Каква е причината за некоректния изход?

Задача “Производител-консуматор” - решение

```
semaphore mutex=1;

Producer:       for (;;) {
                  generate_data;
                  P(mutex);
                  fill_the_buffer;  // CS
                  V(mutex);
                }
Consumer:       for (;;) {
                  P(mutex);
                  read_from_buffer;  // CS
                  V(mutex);
                  consume_data;
                }
```

Задача “Производител-консуматор” - решение

Какъв ще е резултата?

- a a a a a a a b b c d d d d d d d e e ...
- b f p z
- z

Пример от реалността

Производител:

който ще черпи

Консуматор:

когото ще черпят

Общ ресурс:

Защита на общия ресурс

P(mutex)

V(mutex)

Ситуация 1: консуматорът е наред да се почерпи

Ситуация 1: консуматорът е наред да се почерпи

- **Но чашата е празна!**

Ситуация 2: производителят е наред да налее

- **Но чашата е пълна!**

- **Има условия, при които да се извършват действията:**

1. **Консуматорът ще консумира ако чашата НЕ Е ПРАЗНА, ако не е - чака**
2. **Производителят ще налива ако чашата НЕ Е ПЪЛНА, ако не е - чака**

Задача “Производител-консуматор”

```
semaphore empty=1, full=0;
semaphore mutex=1;


Producer:       for (;;) {
                  generate_data;
                  P(empty);
                  P(mutex);
                  fill_the_buffer;
                  V(mutex);
                  V(full);
                }
Consumer:       for (;;) {
                  P(full);
                  P(mutex);
                  read_from_buffer;
                  V(mutex);
                  V(empty);
                  consume_data;
                }
```

Проблеми при семафори

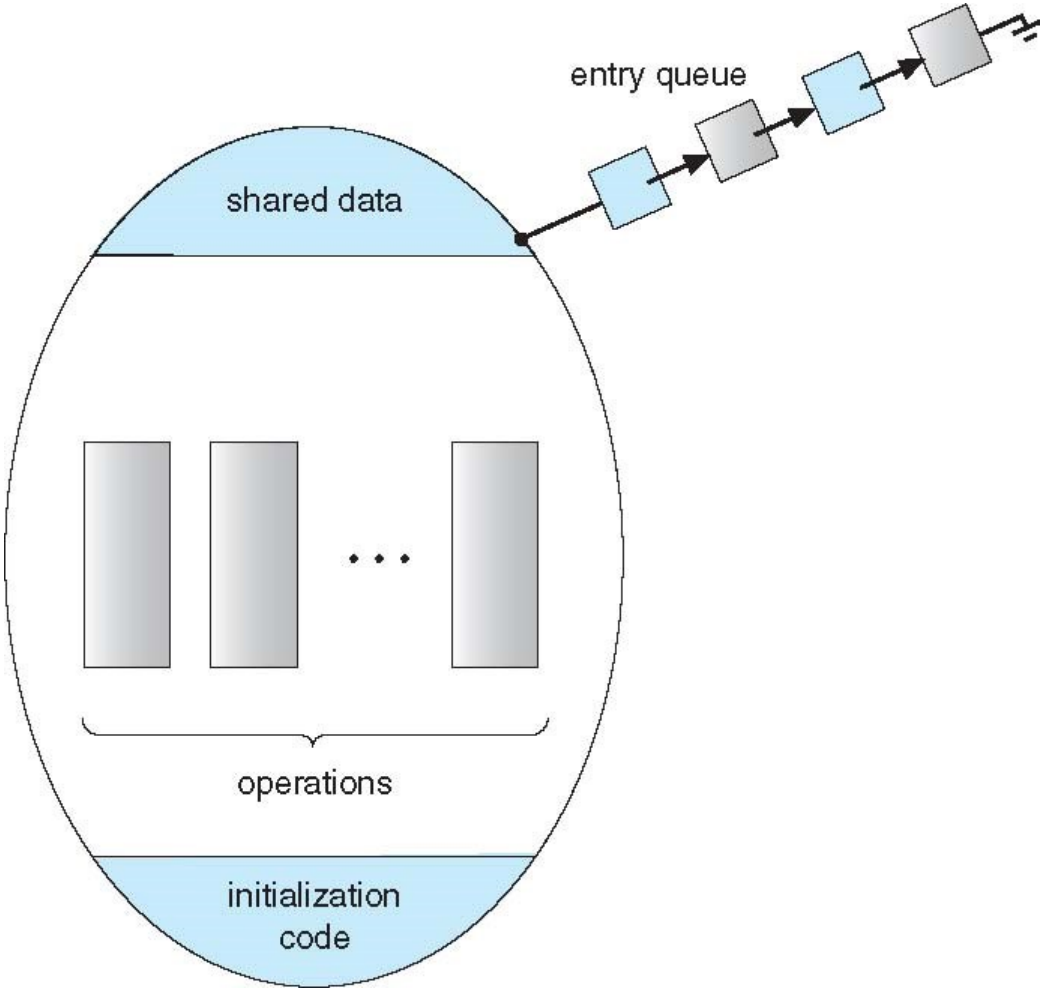
- **Некоректно използване на семафори:**
  - V(S) … P(S)
  - P(S) … P(S)
  - Изпускане на V(S) или P(S)
- **Възможно е настъпване на взаимна блокировка.**
- **Решение: създаване на програмни езикови конструкции от високо ниво.**

Монитори

- Високо ниво на абстракция, предоставящо ефективен механизъм за синхронизация на процеси.
- Мониторът включва данни и взаимоизключващи се операции върху тях.
- Само един процес може да бъде активен в монитора в даден момент
- Взаимното изключване се гарантира от компилатора.

```
monitor monitor-name
{
        // shared variable declarations
        procedure P1 (…) {……}

        procedure Pn (…) {……}

        Initialization code (…) { … }

}
```

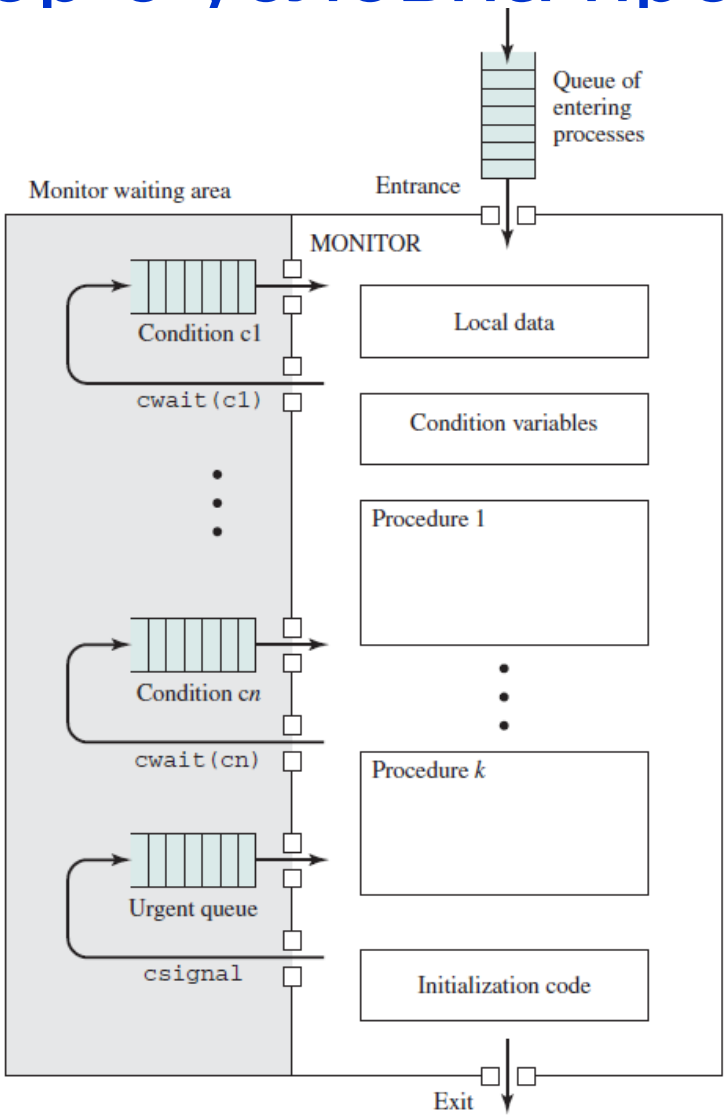
Схематично представяне на монитор



Условни променливи

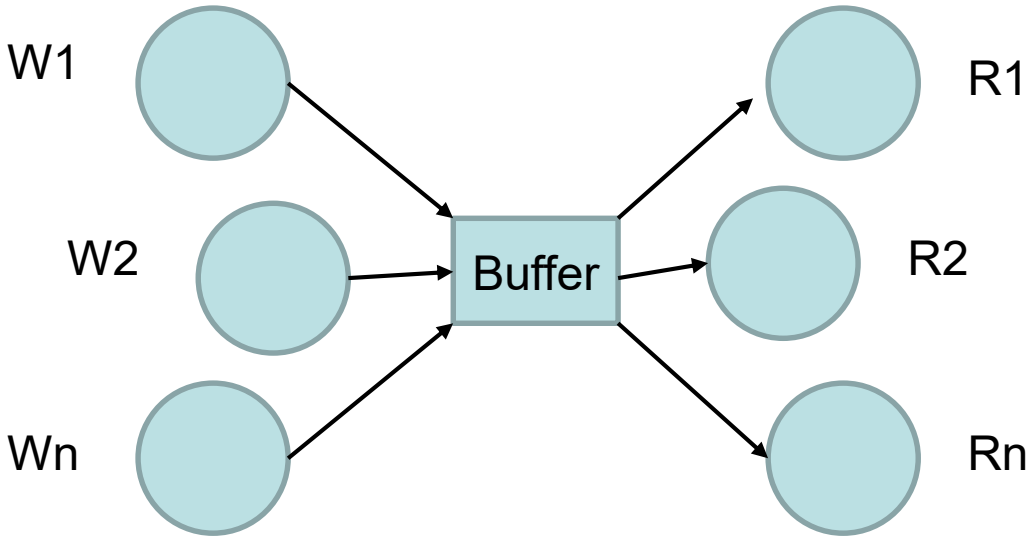
- Необходим е механизъм, позволяващ на процесите на изчакват в монитор.
- Използват се условни променливи (***conditional variables***).
- Условна променлива **`x`** може да бъде достъпна само чрез два оператора:
  - **`x.wait()`** – извикващият процес се блокира, докато друг не изпълни **`x.signal()`**. Мониторът е наличен за използване от друг процес.
  - **`x.signal()`** – възстановява един от процесите (ако има), който е бил извикал **`x.wait()`**. Ако няма блокиран към променливата процес, действието няма ефект върху нея.

Монитор с условна променлива



Задача “Читатели-писатели”

- Наличен е единичен буфер.
- Само един процес-писател може да записва данни в буфера в даден момент.
- Много процеси-читатели могат да четат данните от буфера



Задача “Читатели-писатели”

```
monitor RWmonitor {
    int readercount;
    bool busy;
    condition OKtoread, OKtowrite;

    void startread() {
      if (busy) OKtoread.wait();
      readercount++;
      OKtoread.signal();
    }
    void endread() {
      readercount--;
      if (readercount==0)
         OKtowrite.signal;
    }
    void startwrite() {
      if (busy || (readercount!=0))
         OKtowrite.wait()
      busy=true;
    }
    void endwrite() {
         busy=false;
         if (OKtoread.queue)
                  OKtoread.signal()
         else OKtowrite.signal();
    }
    void init() {
      busy=false;
      readercount=0;
    }
}
```

```
RWmonitor R;

    init();

Writers:
          ....
          R.startwrite();
          write_to_buffer;
          R.endwrite();
          ....

Readers:
          ....
          R.startread();
          read_from_buffer;
          R.endread();
          ....
```

Взаимна блокировка

- Изискване на ресурси.
- Ако ресурс не е наличен, процесът се блокира.
- Взаимна блокировка (***deadlock***) – два или повече процеса чакат безкрайно за настъпване на събитие, което може да бъде инициирано само от някой от тях.
- Безкрайно блокиране (***starvation***) – процес може никога да не бъде премахнат от опашката на семафора, към който е блокиран.

Системен модел

- Системата съдържа ресурси
- Типове ресурси R1, R2, . . ., Rm
  - CPU цикли, обем памет, I/O устройства
- Всеки ресурсен тип R_i има W_i инстанции.
- Процесите използват ресурсите в последователност:
  - Заявка;
  - Използване;
  - Освобождаване.

Условия за deadlock

- Взаимно изключване (*mutual exclusion*);
- Задържане и чакане (*hold and wait*);
- Отсъствие на изтласкване (*no preemption*);
- **Кръгово очакване (*circular wait*).**

- **Възможност**
- **Наличие**

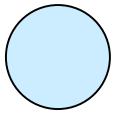
Граф на заемане на ресурси

Множество от върхове V и дъги E:

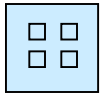
- Множеството V:
  - Всички процеси P = {P1, … Pn);
  - Всички ресурси R = {R1, … Rn);
- Заявка за ресурс - Pi->Rj;
- Използване на ресурс - Rj <- Pi.

Граф на заемане на ресурси

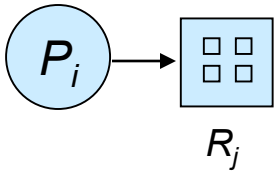
- Процес



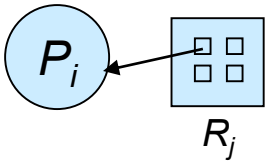
- Ресурсен тип с 4 инстанции



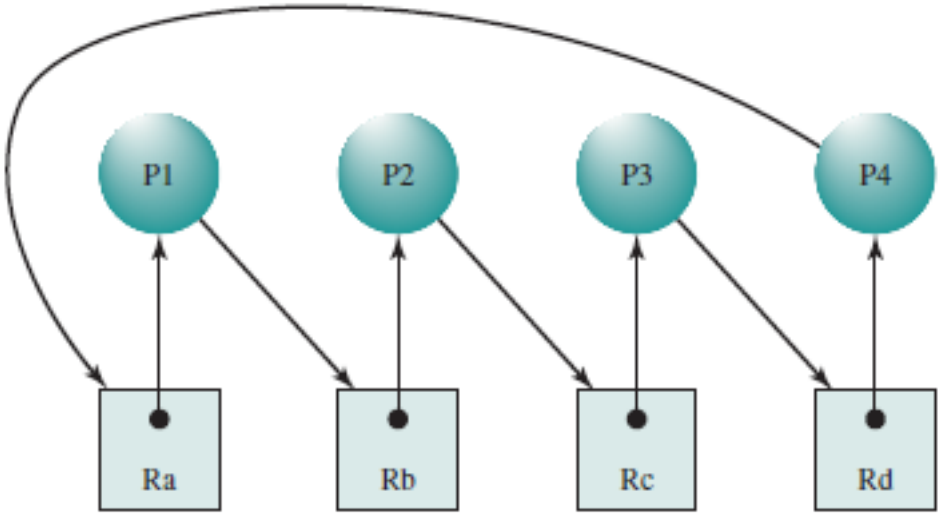
- Процес заявява инстанция на ресурс Rj



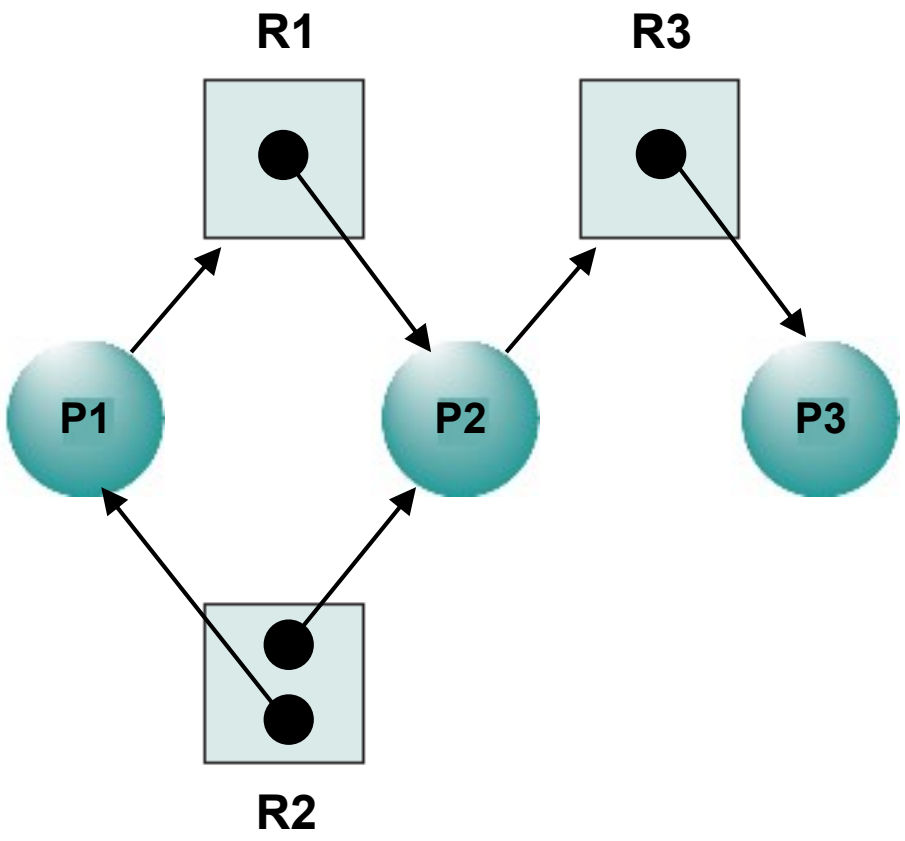
- Процес използва инстанция на ресурс Rj



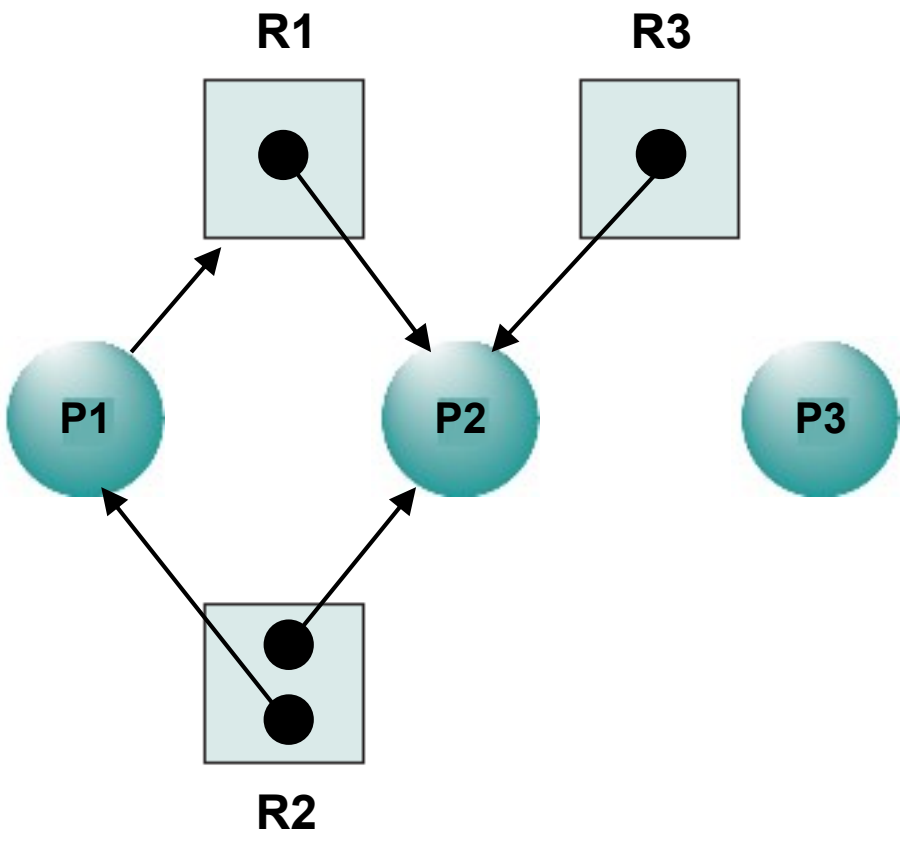
Граф на заемане на ресурси (3)



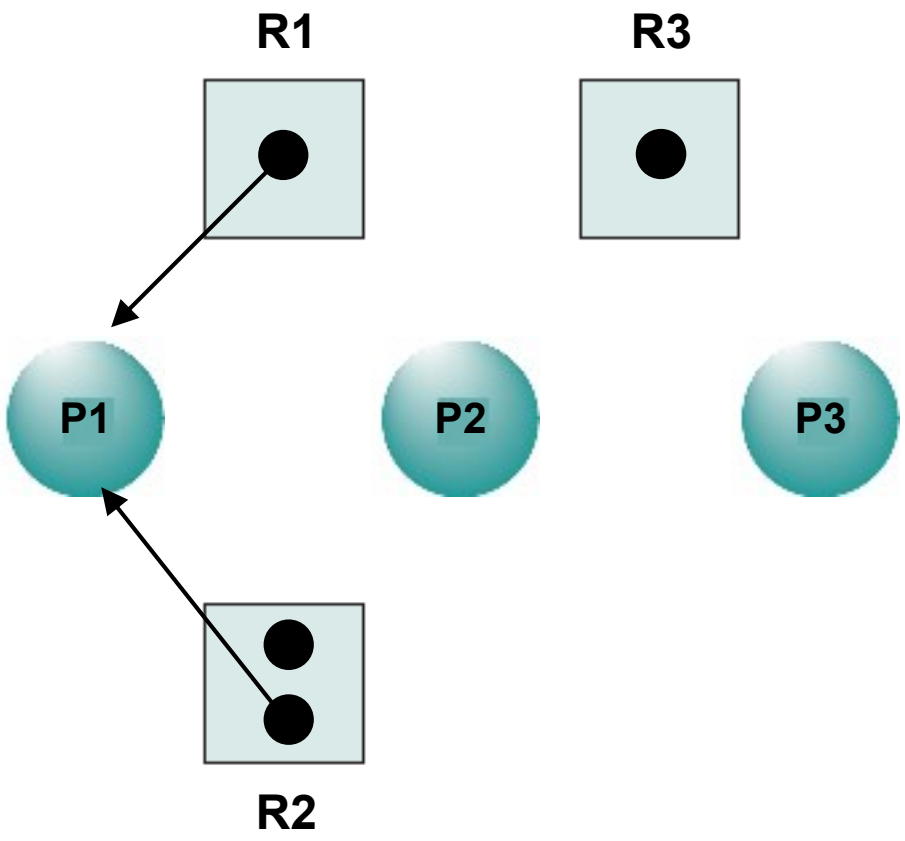
Без наличие на deadlock (1)



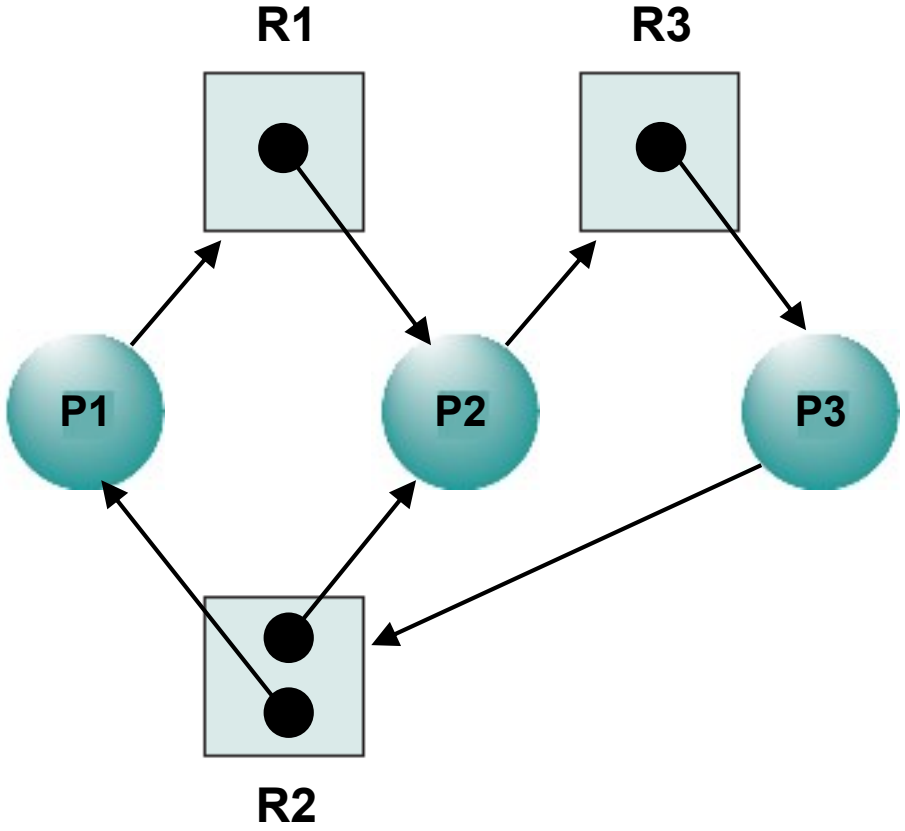
Без наличие на deadlock (2)



Без наличие на deadlock (3)

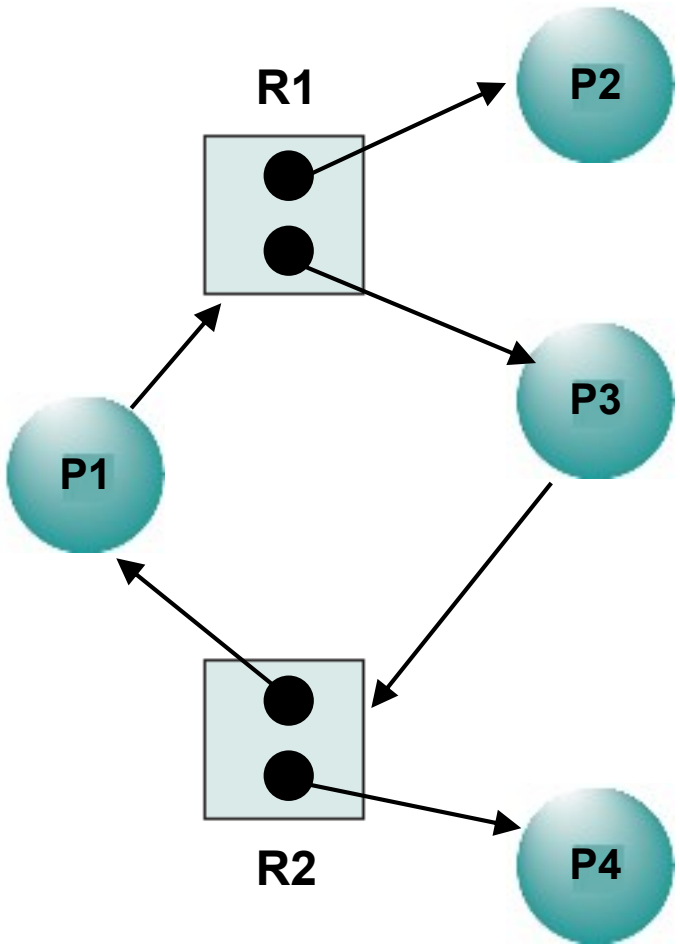


Deadlock



$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$
$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

Наличие на цикъл без deadlock



$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

Граф на заемане на ресурси (3)

Ако графът не съдържа цикли => без deadlock

Ако графът съдържа цикли:

- Ако има само една инстанция за ресурсен тип => **deadlock**;
- Ако има няколко инстанции за ресурсен тип => възможност за deadlock.

Пример за deadlock

```
Process 1:

…
lock(mutex1);
lock(mutex2);
…
use_resource_1;
use_resourse_2;
…
unlock(mutex2);
unlock(mutex1);
…
```

```
Process 2:

…
lock(mutex2);
lock(mutex1);
…
use_resource_2;
use_resourse_1;
…
unlock(mutex1);
unlock(mutex2);
…
```

Решение – “Заключващи йерархии”

```
Process 1:

…
lock(mutex1);
lock(mutex2);
…
use_resource_1;
use_resourse_2;
…
unlock(mutex2);
unlock(mutex1);
…
```

```
Process 2:

…
lock(mutex1);
lock(mutex2);
…
use_resource_2;
use_resourse_1;
…
unlock(mutex1);
unlock(mutex2);
…
```

Въпроси?