Оперативни Системи

Меѓупроцесна комуникација

проф. д-р Димитар Трајанов,
проф. д-р Невена Ацковска,
проф. д-р Боро Јакимовски,
проф д-р Весна Димитрова,
проф. д-р Игор Мишковски,
проф. д-р Сашо Граматиков,
вонр. проф. д-р Милош Јовановиќ,
вонр. проф. д-р Ристе Стојанов,
доц. д-р Костадин Мишев

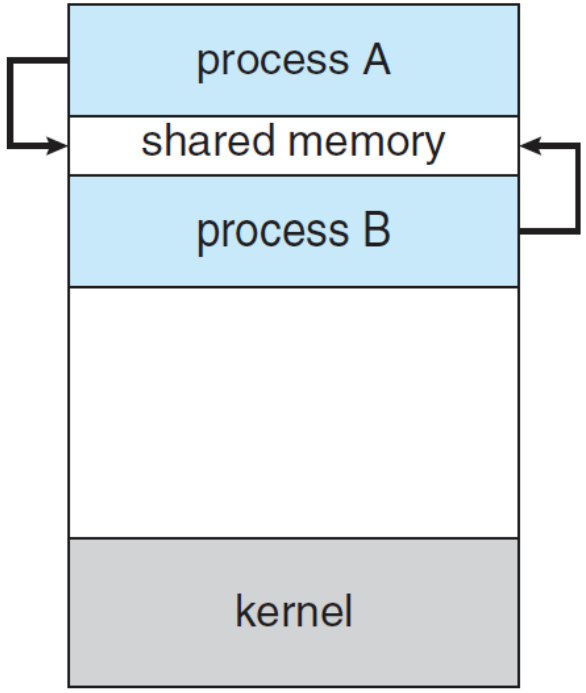
"Ss. Cyril and Methodius" University - Skopje
FACULTY OF COMPUTER
SCIENCE AND ENGINEERING

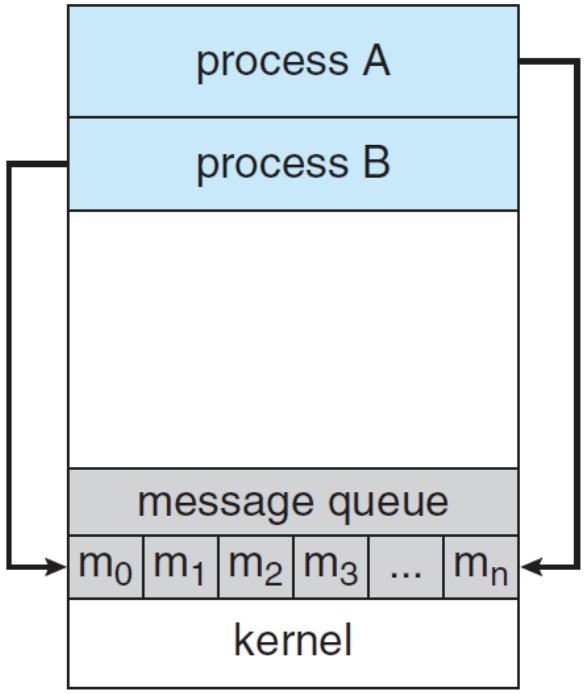
Цел на предавањето

- Меѓупроцесна комуникација (IPC)
  - Критична секција
  - Натпревар на процеси
  - Техники за решавање
    - заклучувачки променливи
    - хардверски решенија
    - семафори
    - монитори
  - Класични проблеми на синхронизација

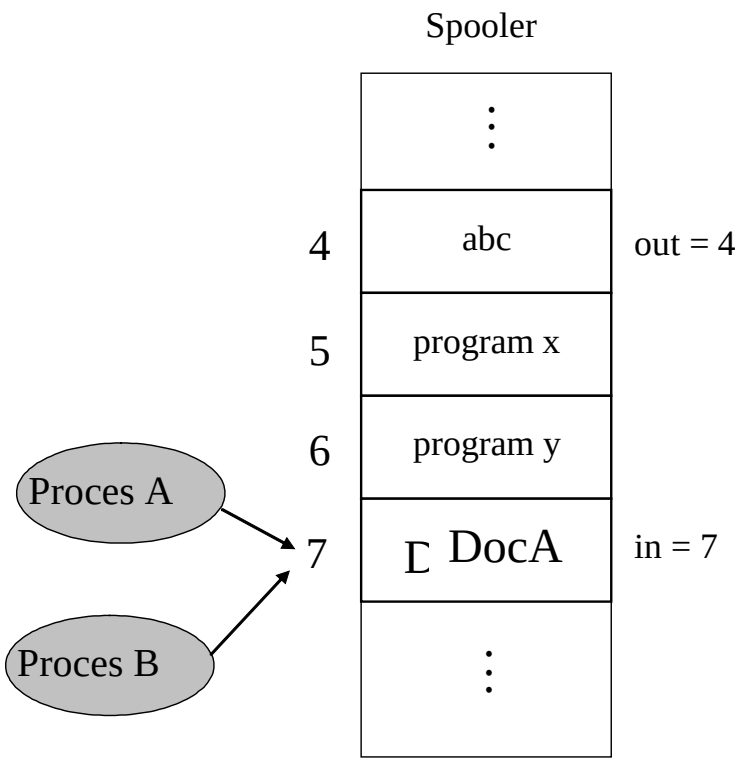
Меѓупроцесна комуникација (Interprocess Communication – IPC)

- Потреба од комуникација меѓу процесите
- Потреба од синхронизација при критичните активности
- Потреба од редослед на операции





Користење заедничка променлива



- **Два процеси А и В сакаат да печатат**
- **Печатењето го прави spooler процес кој управува со редица на документи за печатење преку заеднички споделени променливи**
  - ***in*** **–позиција на последен документ од редицата**
  - **out – позиција на прв документ од редицата**
- **Во моментот кога процесот А ја чита заедничката променлива *in* и ја меморира во локална променлива *sled-slob,* му поминува времето на работа**
- **Доаѓа процесот В, ја чита *in*, добива вредност 7, ја праќа датотеката на таа локација и ја поставува *in* на 8**
- **Процесот А се активира одново и ја праќа датотеката како што му кажува променливата *sled-slob*, на локацијата 7, “газејќи” ја датотеката на процесот В**

ПРОБЛЕМ: процесот В почнува да користи заедничка (shared) променлива пред А да заврши со неа

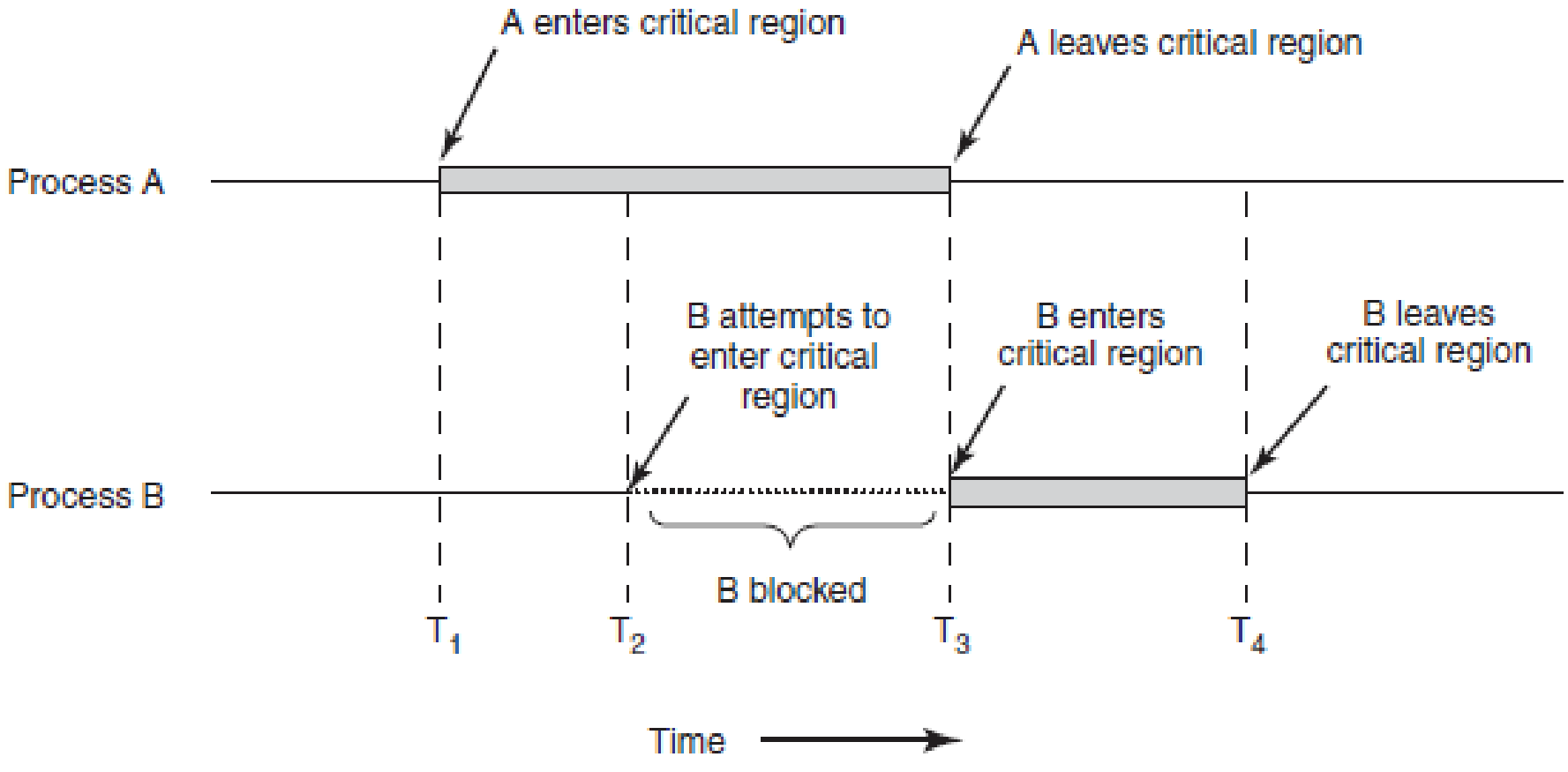
Услови на трка (race conditions)

- Ситуации во кои повеќе процеси читаат или запишуваат во заеднички ресурс (променлива, датотека) и целокупниот резултат зависи од редоследот на извршување на процесите
- Проблемите кои ги предизвикуваат условите на трка тешко се детектираат
- Потребно е да се избегнат по секоја цена
  - Взаемно исклучување на процесите при користење на заеднички ресурси

Критичен сегмент/секција

- Критичен сегмент е дел од програмата каде се пристапува до заедничка меморија (променливи), датотеки, итн.
- Претставува апстракција што се состои од број на последователни програмски наредби кои треба да се извршуваат без прекин и грешки

Критични секции и взаемно исклучување

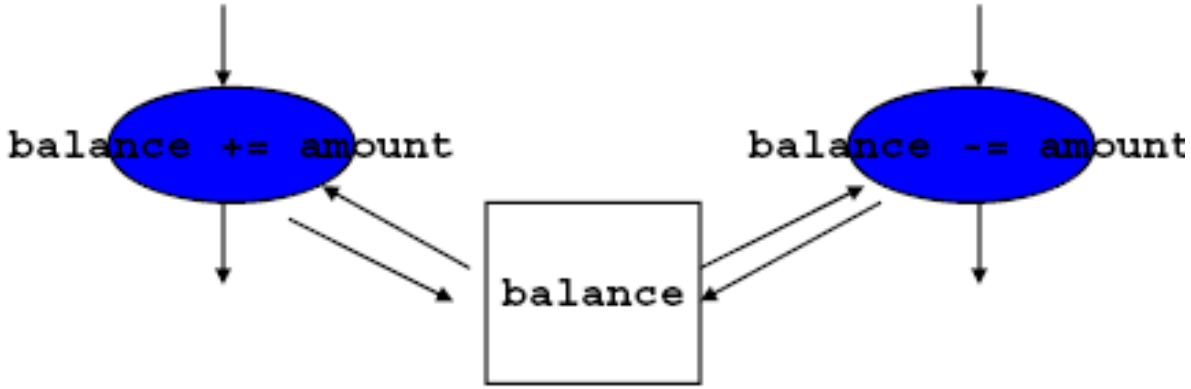


Критична секција

shared double balance;

Код за p1	Код за p2
. . .	. . .
balance = balance + amount;	balance = balance - amount;

Код за p1	Код за p2
load R1, balance	load R1, balance
load R2, amount	load R2, amount
add R1, R2	sub R1, R2
store R1, balance	store R1, balance



Натпревар на процеси

- Постои натпревар на процеси; кој да ја изврши својата критична секција
- Секциите можат да бидат дефинирани од различен код во различни процеси
  - не е лесно да се детектира со статичка анализа
- Без взаемно исклучување, резултатите од повеќекратните извршувања може да се неконзистентни
- Потребен е ОС механизам за корисникот да може да се справи со натпреварите на процеси

Проблем на критични секции

- Даден е систем од n процеси {P0, P1, … Pn-1}
- Секој процес има критична секција (менува вредност на променливи, табели, запишува датотеки кои се заеднички со други процеси...)
- Правило:
  - Кога еден процес работи во својата критична секција, ниту еден друг процес нема дозвола да извршува во својата критична секција
- Треба да се дизајнира протокол за кооперација на процесите

Генерална структура на Pi

- Секој процес мора да бара дозвола да влезе во критичната секција
- Критичната секција може да е следена од излезна секција

```
do {
        entry_section
        critical_section
        exit_section
        reminder_section
}
```

Можни ОС механизми

- Оневозможи прекини
- Софтверски решенија – заклучување променливи (брави, locks), семафори
- …

Критериуми за имплементација

- Секоја имплементација мора да биде:
  - **коректна**: само еден процес може да ја извршува критичната секција во еден момент
  - **ефикасна**: влегувањето и излегувањето од критичната секција мора да биде брзо
  - **флексибилна**: добрата имплементација дозволува максимална конкурентност и има минимален број на ограничувања

Коректност на решението

- Коректното решение мора да ги има следниве својства:
- **Взаемна исклучивост**: ако еден процес работи во критична секција, ниеден друг не смее да работи во својата критична секција (најмногу еден)

- **Прогрес**: ако нема процес што моментално работи во критичната секција, а има процеси што сакаат да работат во критичната секција, тогаш еден од нив добива дозвола (најмалку еден)

- **Ограничено чекање**: мора да има граница на бројот на дозволи која ја добиваат други процеси за влез во својата критична секција пред да биде услужен даден процес што има барање за влез во критична секција (ниту еден процес не смее да чека бесконечно долго за влез во својата критична секција)

1. Заедничка заклучувачка променлива – работно чекање

Алгоритам 1

- Процесите (P_0 и P_1) делат заедничка променлива **turn.**
- Кога turn==i, тогаш процесот P_i има дозвола да влезе во критичната секција. Другиот процес е P_j за j==1-i;

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Погодности и проблеми

- Решението овозможува само еден процес да биде во едно време во критичната секција.
- **Не задоволува прогрес**: потребна е стриктна алтернација на процесите кои се одвиваат во критична секција.
- Проблем е ако едниот процес е многу поспор од другиот

Чекор	Процес 0	Процес 1
1		turn = 0
2	wait for turn = 0	noncritical_region (enter)
3	critical_region (enter)	
4	critical_region (finish)	
5	turn = 1	
6	noncritical_region (enter)	
7	noncritical_region (finish)	
8	wait for turn = 0 (правило 2 е прекршено)	

Алгоритам 2

- Алгоритам 1 не содржи доволно информации за состојбите на секој процес, туку само на кој процес му е дозволено да влезе во критичен сегмент
- Се воведува низата boolean flag[P1..P2]
- Ако flag[Pi] е true, i-тиот процес Pi е подготвен да влезе во критичната секција
- i-тиот процес проверува дали j–тиот процес НЕ е спремен да влезе во неговиот критичен сегмент
- Ако и flag[Pj] е true, тогаш Pi чека додека flag[Pj] стане false, па тогаш Pi ќе влезе во критичниот сегмент

Решение: заклучување

АЛГОРИТАМ 2

Flag[P1..P2]

Process 1

```
Flag[P1] = true;
while(Flag[P2])
    ; // wait
/* enter C.S. */
/* leave C.S. */
Flag[P1] = false;
……..
```

Process 2

```
Flag[P2] = true;
while(Flag[P1])
    ; // wait
/* enter C.S. */
/* leave C.S. */
Flag[P2] = false;
……..
```

Взаемно исклучување со работно чекање

Проблем:

deadlock: (Flag[P_1]=true , Flag[P_2]=true), бесконечен циклус

Правилото 3 е прекршено.

Алгоритам 3

- Со комбинација на претходните два алгоритма се добива алгоритам што ги исполнува 3те барања за коректност

- За да влезе во критичен сегмент Pi го сетира flag[Pi] на true, и поставува turn=Pj, со што означува дека ако другите процеси сакаат да влезат во критичен сегмент, можат

- Оној процес што го сетира flag-от и добие своја вредност во променливата turn, влегува во критичен сегмент

Решение: комбинација

АЛГОРИТАМ 3

Flag[P1..P2], turn

Process 1

```
Flag[P1] = true;
turn = P2;
while(Flag[P2] and turn == P2)
    ;
/* enter C.S. */
/* leave C.S. */
Flag[P1] = false;
……..
```

Process 2

```
Flag[P2] = true;
turn = P1;
while(Flag[P1] and turn == P1)
    ;
/* enter C.S. */
/* leave C.S. */ / /
Flag[P2] = false;
……..
```

Process 1 Asembler

```
store Flag[P1], 1
load $5,P2
store turn,$5
WHILE:
Load $5, P2
Load $6, turn
Load $7, Flag[P2]
Xor $8, $7,1
Xor $9, $5,$6
OR $8, $8,$9
Be $8,$0, WHILE
/* enter C.S. */
/* leave C.S. */
store Flag[P1], 0
……..
```

Ова е идеја на Peterson-овото решение

(модерна верзија на Dekker-овиот алгоритам

Петерсоново решение

```
#include <prototypes.h>
#define FALSE 0
#define TRUE 1
#define N 2                                   // number of processes
turn int turn, interested[N]                  // all values initially 0
void enter_region(int process)
{
      int other;                              // number of the other process
      other = 1 - process;
      interested[process] = TRUE;             // show that you are interested
      turn = process;                          // set the flag
      while ( turn == process && interested[other] == true) ; // wait
}
void leave_region(int process) { // process who is leaving (0 or 1)
   interested[process] = FALSE; // indicate departure from critical region
}
```

Петерсоново решение (пример)

Process = 0

```
interested[0]=interested[1]=FALSE
int turn;
int interested[2];

void  enter_region(...){
    int  other = 1;
    interested[0] = TRUE;
    turn = 0;
    while (turn == 0 &&
           interested[1] == TRUE);
}

void  leave_region(...){
    interested[0] = FALSE;}
```

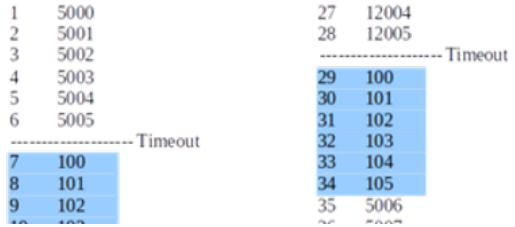
Process = 1

```
interested[0]=interested[1]=FALSE
int turn;
int interested[2];

void  enter_region(...){
    int  other = 0;
    interested[1] = TRUE;
    turn = 1;
    while (turn == 1 &&
           interested[0] == TRUE);
}

void  leave_region(...){
    interested[1] = FALSE;}
```

2. Заклучување – хардверска поддршка

- Исклучување прекини
- Специјални машински инструкции

2а. Исклучување на прекините



- За системи без нишки (threads), прекините се единствен извор на проблеми поврзани со конкурентност и синхронизација
- Едноставно решение:
  - Исклучи ги прекините пред почетокот на критичниот сегмент
  - Вклучи ги прекините по критичниот сегмент
  - со исклучени прекини, нема временско ограничување за процесот
- За ОС исклучувањето на хардверските прекини е опасно: може да доведе до одложување на одговор на важни настани
- Не е соодветно за генерална техника за кориснички процеси
  - Критичните сегменти мораат да бидат куси
  - Не е соодветно кај системи со повеќе процесори (јадра)

Пример: исклучување прекини

```
shared double balance;
```

```
Код за p1
disableInterrupts();
balance = balance + amount;
enableInterrupts();
```

```
Код за p2
disableInterrupts();
balance = balance - amount;
enableInterrupts();
```

- Прекините може да бидат оневозможени произволно долго
- Ние сакаме само p_1 и p_2 да не се мешаат меѓу себе; ова ги блокира сите p_i
- Да искористиме заедничка брава “lock” променлива

2б. Специјални инструкции

- Специјалните машински инструкции
  - се изведуваат во **единечен** инструкциски циклус
  - не се подложни на меѓусебно мешање на инструкциите
  - Се регулира достап до мемориска локација
    - читање и запишување
    - читање и тестирање

Test and Set Инструкција

- се изведува атомично, во еден циклус, без прекини

```
enter_region:
        TSL REGISTER,LOCK          | copy lock to register and set lock to 1
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was non zero, lock was set, so loop
        RET | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET | return to caller
```

Test and Set инструкција - реализација

- Оваа инструкција не може да биде прекината – никој не може да пристапи до меморијата додека таа се извршува

```
boolean testset (int i) {
        if (i == 0) {
            i = 1;
            return true;
        }
        else {
            return false;
        }
    }
```

Замена на вредности

- Exchange инструкција
- замена меѓу регистар и мемориска локација
- атомична операција, во една инструкција

```
enter_region:
        MOVE REGISTER,#1          | put a 1 in the register
        XCHG REGISTER,LOCK        | swap the contents of the register and lock variable
        CMP REGISTER,#0           | was lock zero?
        JNE enter_region          | if it was non zero, lock was set, so loop
        RET                       | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0              | store a 0 in lock
        RET                       | return to caller
```

XCHG реализација

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

Машински инструкции за взаемно исклучување

- Предности
  - можат да се искористат за било кој број процесори кои може да делат работна меморија
  - едноставни се за работа и за верификација
  - можат да се искористат за неколку критични секции

Машински инструкции за взаемно исклучување

- Неповолности
  - зафатеното чекање троши процесорско време
  - можно е изгладнување кога некој процес излегува од критична секција, а повеќе од еден процес чекаат на неа

Општ проблем

- Овие пристапи (работно чекање) непотребно го трошат процесирачкото време чекајќи да добијат дозвола за влез во критичниот сегмент
- Пример. 2 процеса,
  - H – со висок приоритет (веднаш му се доделува CPU кога е во спремна состојба)
  - L – со низок приоритет
  - Кога L е во критичен сегмент, H станува спремен и му се доделува CPU . Тогаш H почнува со работно чекање, но L не може да излезе од критичниот сегмент бидејќи H работи (иако чека).
  - Со тоа H е во бесконечен циклус

sleep(), wakeup()

- Подобро е да се користат некои системски повици кои го блокираат процесот на кој не му е дозволен влез во критичниот сегмент
- На пример:
  - sleep() да го суспендира процесот додека некој друг не го разбуди со wakeup()
  - wakeup() има еден параметар, процесот кој треба да се разбуди
- Системските повици од овој тип водат кон решенија со блокирање

Проблем на ограничен бафер

- Важен поради: мрежни интерфејси, В/И контролери, размена на пораки итн.

- Кружен бафер со ограничен капацитет
- Пристап: Додавање на крајот, земање од почетокот
- Модел на производувач/потрошувач со: N : број на места, count : број на пополнети

Решение со класично блокирање

```
#include <prototypes.h>
#define N 100 // number of slots
int count 0;
void producer (void) {
    int item;
    while (TRUE) {
    produce_item(&item);  // next item
    if (count ==N) sleep(); // full?
    enter_item(item);      // put item
    count = count + 1;
    if (count == 1)
    wakeup(consumer);    // empty?
} }
```

```
void consumer (void) {
    int item;
    while (TRUE) {
    enter
    if (count == 0) sleep(); // empty?
    remove_item(&item);  // take item
    count = count - 1;
    if (count == N-1)
    wakeup(producer);      // full?
    consume_item(item); // print item
} }
```

Во ова решение променливата count не е заштитена!

Реализација

Producer	Consumer
	count=0, read_count
insert_item()	
count++	
wakeup(consumer)	**Problem: consumer is not sleeping (signal is lost)**
wait for next item	
	count=0 => sleep
insert_item()	
…	
count =N => sleep	**Both processes sleep forever!**

3. Семафори

- Решенијата со работно чекање тешко се генерализираат за покомплексни проблеми
- Се користат **семафори**
- Семафорот се состои од:
  - Променлива value
  - Листа на процеси L
  - Кон семафорот се пристапува со
    - Функција wait (down, sleep) – блокира процес
    - Функција signal (up, wakeup) – буди процес
- Пример: Semaphore s(5);

Функционалност од семафори

- Процесот ја повикува **wait** во критичен момент и во зависност од вредноста на променливата **value** во семафорот, добива дозвола за продолжување или станува блокиран.
- Процесот ја повикува **signal** по поминување на критичниот сегмент и така ослободува еден процес да помине низ семафорот.
  - Ослободениот процес поминува во спремна состојба и се сместува во редицата со спремни процеси (CPU се доделува според алгоритам за распределување)

Својства

- Променливата **value** одредува колку процеси можат симултано да работат (пропусна моќ на семафорот, т.е. колку процеси може истовремено да се опслужат – пример 3 печатачи).
- Овие кодови се извршуваат атомично, т.е додека се одвива операцијата со семафорот, ниту еден друг процес нема пристап до семафорот

Имплементација на семафорите

```
typedef struct{
    int value;
    struct process *L;
} semaphore;
```

```
void wait(semaphore S) { // sleep, down
S.value --;
if (S.value < 0) {
    add process to S.L;
    block();
    }
}
```

```
void signal (semaphore S) {//wakeup, up
S.value++;
if (S.value ≤ 0) {
    Remove a process P from S.L;
    wakeup(P);
    }
}
```

Имплементација на семафорите на системско ниво

- Семафорите се имплементираат во јадрото (kernel)
  - Вредноста на семафорот се чува во табела во меморијата на јадрото (kernel memory)
  - Секој семафор се идентификува со број кој соодветствува на неговата позиција во табелата.
  - Постојат системски повици за креирање и бришење на семафори, како и за извршување на операциите wait и signal
  - Операциите се извршуваат атомично
    - Кај процесори со едно јадро се оневозможуваат прекините
    - Кај повеќејадрени процесори се користат атомични инструкции за читање и тестирање (пр. TSL)

Семафор како генерална синхронизирачка алатка

- **Бројачки семафори** – целобројни вредности - неограничени
- **Бинарни семафори** – целобројни вредности меѓу 0 и 1; полесни за имплементација
  - попознати како mutex брави
- Бројачкиот семафор S може да се имплементира како бинарен

Користење семафори

- n процеси делат ист семафор mutex.
- Секој Pi е организиран:

```
do {
        wait(mutex);
        critical_section
        signal(mutex);
        non critical section
} while(1)
```

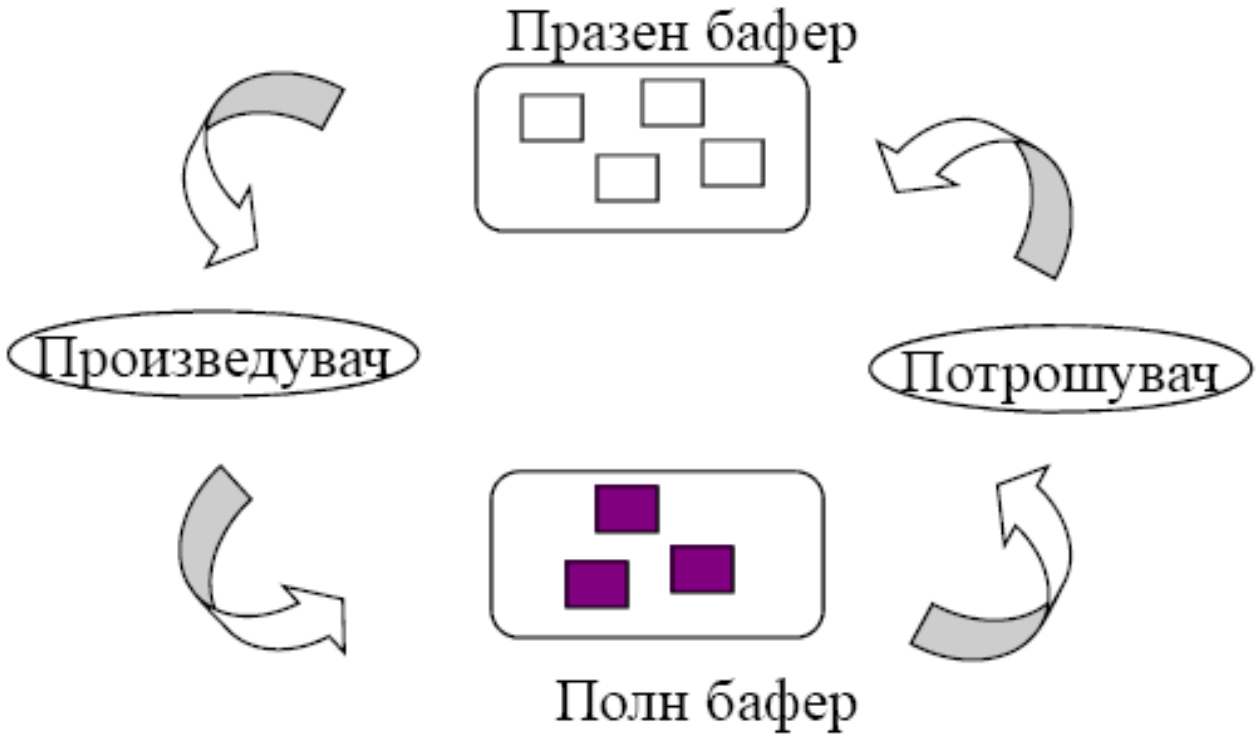
Пример

- Два процеса кои работат конкурентно:
  - P1 има наредби S1, а P2 има S2
- Нека наредбите S2 треба да се извршат откако завршиле наредбите S1, без разлика кој процес го добил прв CPU
- P1 и P2 користат заеднички семафор sync=0 во својот код:

```
S1;                          wait(sync);
signal(sync); //vo P1        S2; //vo P2
```

- Бидејќи sync==0, P2 ќе ги изврши S2 само откако P1 ќе повика signal(sync); што е по S1

Произведувач - потрошувач



Произведувач – потрошувач: Семафори

```
#include <prototypes.h>
#define N   100     // number of slots
typedef int semaphore;
semaphore mutex=1, empty=N, full=0;

void producer (void)  {
    int item;
    while (TRUE)  {
     produce_item(&item);   // next item
     Down (&empty);  //  full? decrement empty
     Down (&mutex);  // enter cr. segment
     enter_item (item);
     Up (&mutex);   // exit cr. segment
     Up (&full);        // increment full
     }   }
```

```
void consumer (void)  {
    int item;
    while (TRUE)  {
     Down (&full);  //  full? decrement
     Down (&mutex);  // cr. segment
     remove_item(&item);   // take item
     Up (&mutex);   // cr. segment
     Up (&empty);    //increment empty
     consume_item(item);   // print item
     }   }
```

4 Конструкции од виши програмски јазици

- Доаѓаат како дел од програмски јазик, односно мора да имаат поддршка од компајлерот
- Претпоставка дека процесите се состојат од локални податоци и код што оперира над тие податоци
- Локалните податоци може да се достапат само преку програмата што е енкапсулирана во самиот процес ...т.е. еден процес не може директно да достапи кон локални податоци на друг
- Процесите може да делат глобални податоци

Монитори

- Множество на процедури, променливи и податочни структури групирани во специјален вид модули
- Процесите можат да го повикаат мониторот во секој момент, но не можат директно да пристапат до внатрешните податочни структури на мониторот со процедури декларирани надвор од мониторот

Опис на монитори

- Монитор е конструкција многу слична на денешните објекти во програмските јазици како C++, Java ...
- Се состои од:
  - Општи променливи (a, b, c)
  - Методи ($fn_1(\ldots)$, $fn_2(\ldots)$; $fn_m(\ldots)$;)
  - Иницијален код (конструктор)
  - Условни променливи (x,y,z)

```
Monitor M {
    int a, b, c ;
    condition x, y, z ;
    fn1(…);
    fn2(…);
    …
    fnm(…);
    { init code}
}
```

Семантика

- Само еден метод (процес) во мониторот може да биде активен во еден момент - взаемно исклучување
  - Ако еден процес повика процедура од монитор додека друг процес извршува некоја од процедурите, тој процес се блокира и се става во редица на чекање (мониторот има редица на процеси кои чекаат да повикаат процедура)
- Програмерот **не треба експлицитно да програмира ограничувања заради синхронизација** (тоа го врши компајлерот)
- Со додавање на **сите критични сегменти како процедури (функции) на мониторот**, нема да има конкурентно извршување на процесите во критичните сегменти

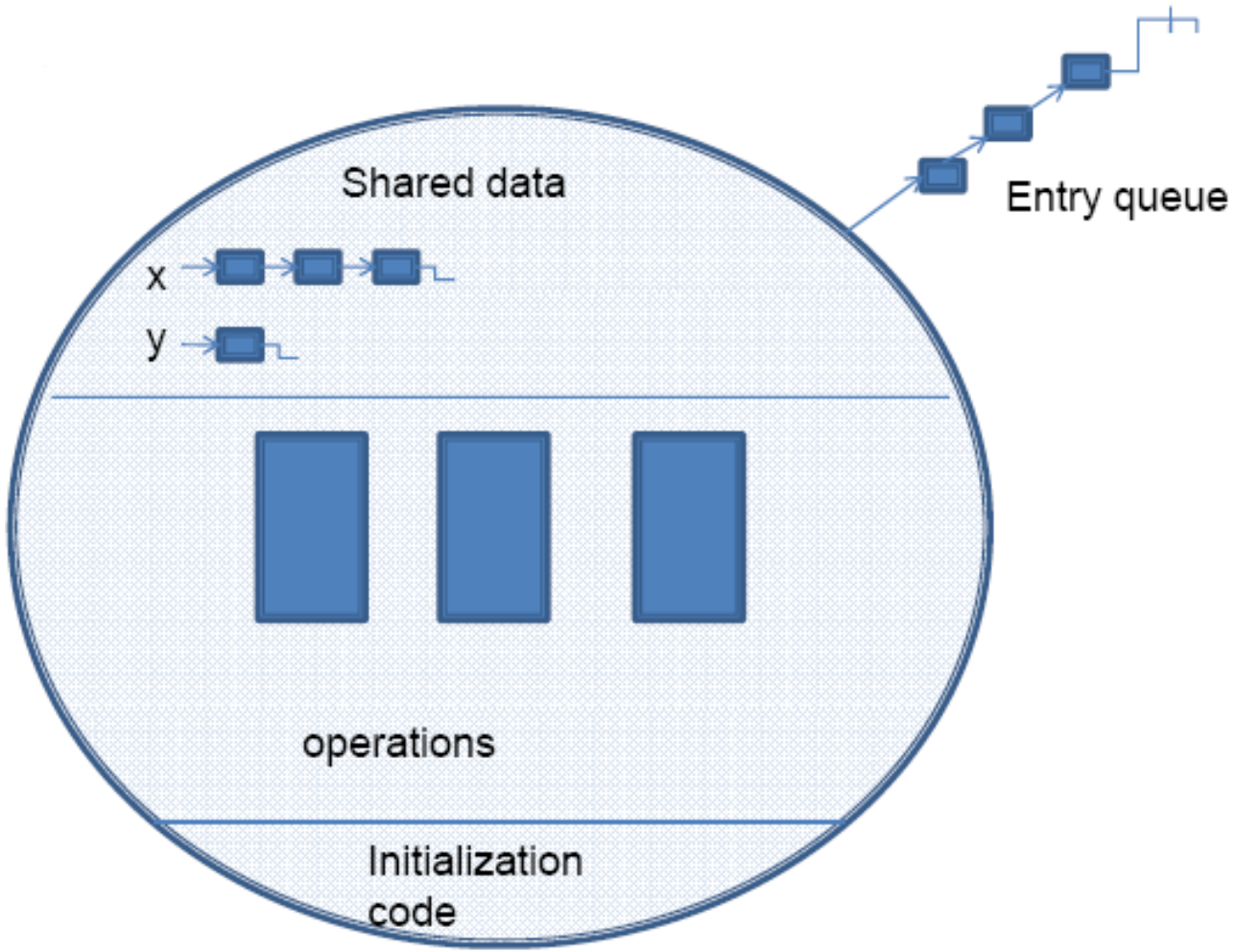
Условни променливи

- Со додавање на условни променливи, се збогатува управувањето со процесите
- Условните променливи имаат две операции:
  - **x.wait** - процесот кој ја повикува оваа операција блокира (чека во ред), се додека некој друг метод (процес) не ја сигнализира променливата со ...
  - **x.signal** - деблокира некого што чека на променливата (... но, со тоа би имале 2 активни процеси во мониторот...)
- Секоја условна променлива има редица на процеси блокирани со wait кои чекаат да бидат разбудени со signal

Користење условни променливи

- Процесот кој повикува wait во рамките на метод на мониторот, го ослободува мониторот за други процеси да можат да извршуваат методи од мониторот
- Процесот кој ја продуцирал операцијата signal, го напушта мониторот (се блокира), а се активира процес кој чека на условната променлива x.
- Ако условната променлива е сигнализирана, а никој не чека на неа, сигналот се губи
  - Wait мора да постои пред signal

Монитор


Shared data
x
y
Entry queue
operations
Initialization
code

Пример

*fn*1(…)
…
x.wait // P1 blocks

// P1 resumes
// P1 finishes

*fn*4(…)
…
x.signal // P2 blocks

// P2 resumes

Задача 1

Претпоставете дека имате **три конкурентни нишки** во рамки на еден процес кои ги извршуваат процедурите B, A, C.

Наведете ја вредноста на **глобалната променлива x** (*внесете цел број*), како и соодветната секвенца на извршување (*внесете ги имињата на процедурите по редослед на завршување без запирка, пример за коректен внес: BAC*), **откако ќе завршат трите нишки**.

```
#include <prototypes.h>
typedef int semaphore;
semaphore sA=1, sB=0, sC=0;
int x = 0;
void A()
{
      Down(&sA);
      x+=2;
      Up($sB);
}
```

```
void B()
{
      Down(&sB);
      x-=1;
      Up($sC);
}
void C()
{
      Down(&sC);
      x+=2;
      Up($sA);
}
```

Вредноста на променливата x е

[ ]

, а секвенцата на извршување е

[ ]

Задача 1 - решение

Претпоставете дека имате **три конкурентни нишки** во рамки на еден процес кои ги извршуваат процедурите B, A, C.

Наведете ја вредноста на **глобалната променлива x** (*внесете цел број*), како и соодветната секвенца на извршување (*внесете ги имињата на процедурите по редослед на завршување без запирка, пример за коректен внес: BAC*), **откако ќе завршат трите нишки**.

```
#include <prototypes.h>
typedef int semaphore;
semaphore sA=1, sB=0, sC=0;
int x = 0;
void A()
{
      Down(&sA);
      x+=2;
      Up($sB);
}
```

```
void B()
{
      Down(&sB);
      x-=1;
      Up($sC);
}
void C()
{
      Down(&sC);
      x+=2;
      Up($sA);
}
```

Вредноста на променливата x е [3], а секвенцата на извршување е [ABC].

Задача 2

Со која најмала вредност треба да се иницијализира семафорот `**s**` во кодот подолу за да нема **deadlock**, ако две различни нишки (threads) T1 и T2, вршат повикување на функциите дефинирани подолу и тоа по следниот редослед:

T1: повикува "A B C A B"
T2: повикува "C B A A B"

Дефиниција на фунцкиите A, B и C е следнава:

```
...
semaphore s=   ;

void A() {
  wait(&s);
  ...
}
```

```
void B() {
  signal(&s);
  ...
}
void C() {
  wait(&s);
  ...
}
...
```

Вредноста на семафорот по завршување на двете нишки ќе биде

.

3 6 4 0 2 5 1

Задача 2 - решение

Со која најмала вредност треба да се иницијализира семафорот `**s**` во кодот подолу за да нема **deadlock**, ако две различни нишки (threads) T1 и T2, вршат повикување на функциите дефинирани подолу и тоа по следниот редослед:

T1: повикува "A B C A B"
T2: повикува "C B A A B"

Дефиниција на фунцкиите A, B и C е следнава:

```
...
semaphore s=[4] ;

void A() {
  wait(&s);
  ...
}
```

```
void B() {
  signal(&s);
  ...
}
void C() {
  wait(&s);
  ...
}
...
```

Вредноста на семафорот по завршување на двете нишки ќе биде [2] .

[3] [6] [0] [5] [1]

Прашања?