Паралелно програмиране

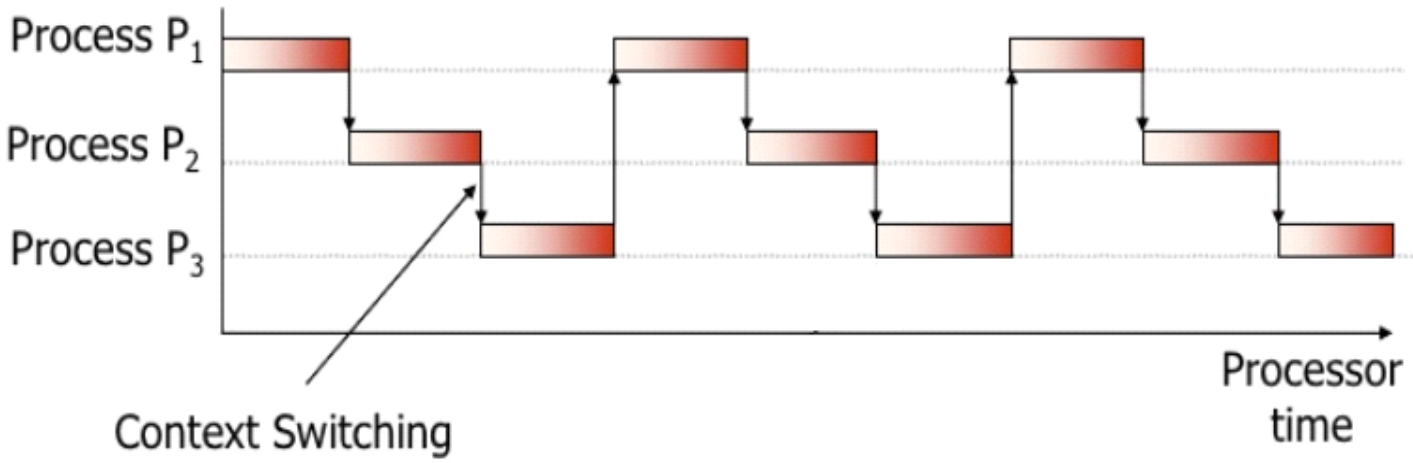
Процес

- Последователен процес е работа извършвана от последователен процесор при изпълнение на програма с нейните данни;
- Процесът е двойката „процесор-програма“, при изпълнение.

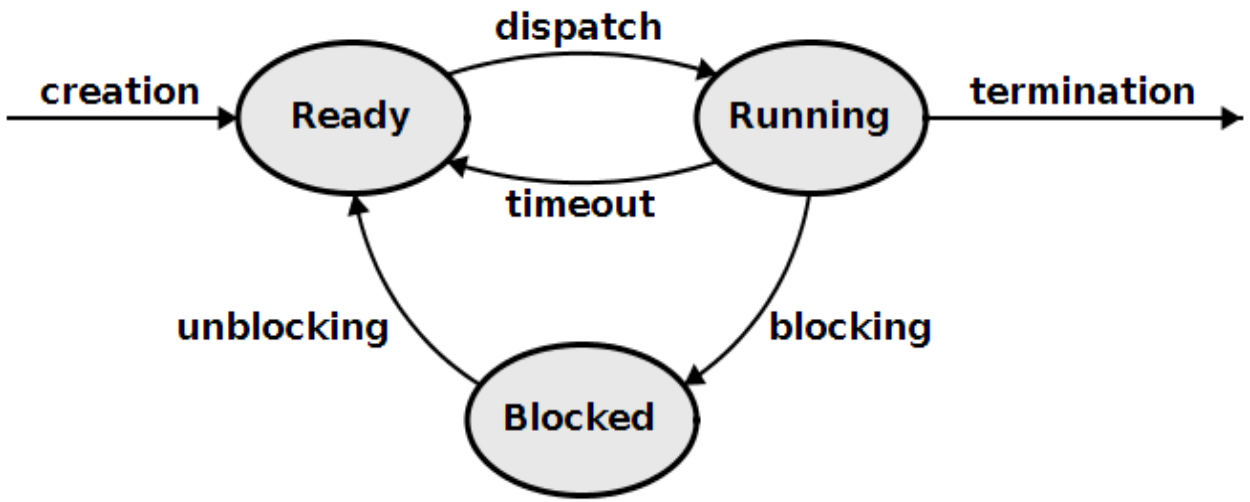
Процес

- Стек;
- Текущи стойност на брояч на команди;
- Стойности на регистри;
- Структури от данни на ОС.

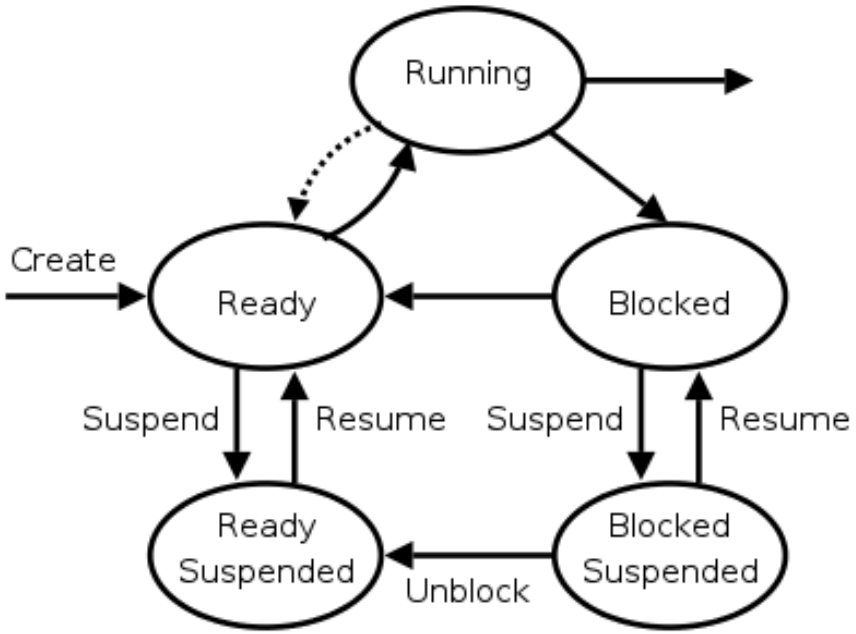
Достъп до процесора (едно ядро)



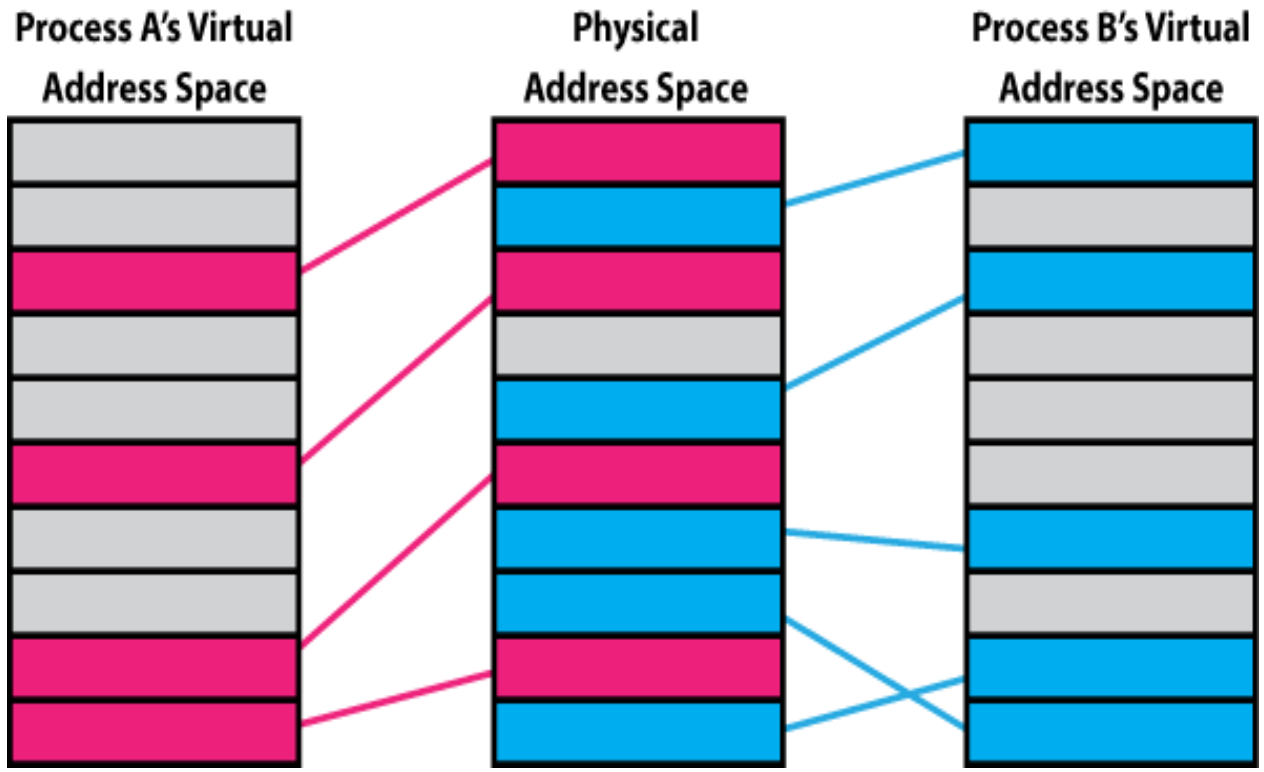
Състояние на процеса



Състояние на процеса



Достъп до паметта



Паралелни процеси

- Изпълнението им се припокрива във времето.

Според достъпа до ресурси

- Независими;
- Взаимодействащи:
  - Конкуриращи;
  - Кооперираща.

Критични ресурси

- Ресурс до който се извършва достъп от паралелни процеси

Критични ресурси - правила

- Само един процес може да използва критичен ресурс в даден момент;
- Ако няколко процеса изискват достъп до критичен ресур, той трябва да бъде предоставен в крайно време;
- Ако процес получи ресурс, той трябва да го освободи в крайно време.

Пр: достъп до брояч

- П1: c++;
  LDAA c;
  INCA;
  STAA c;
- П2: c--;
  LDAB c;
  DECB;
  STAB c;

LDAA c	(A=4);
INCA	(A=5);
LDAB c	(B=4);
DECB	(B=3);
STAA c	(c=5);
STAB c	(c=3);

Критична секция

- Област от процеса, в която той работи с критичен ресурс.

Критична секция - правила

- Само един процес може да се намира в критичната си секция в даден момент;
- Процес може да остане в критичната си секция за крайно време;
- Процес трябва да може да влезе в критичната си секция в крайно време;
- Процес намиращ се извън критичната си секция не може да пречи на други да влязат в своите.

Синхронизация - програмно решение

- Блокировка на паметта - операциите за четене и запис представляват критични секции реализирани на апаратно ниво.

- Изпълнение на команда - изпълнението на машинна команда не се прекъсва.

Пр1:

```
void Program1()
{
    while (turn != 1) { };
    // critical section
    turn = 2;
}
```

```
void Program2()
{
    while (turn != 2) { };
    // critical section
    turn = 1;
}
```

Пр1:

```
void Program1()
{
    while (turn != 1) { };
    // critical section
    turn = 2;
}
```

```
void Program2()
{
    while (turn != 2) { };
    // critical section
    turn = 1;
}
```

- Стриктно редуване;
- Всеки процес определя кога стартира следващия.

Пр2:

```
bool turn1, turn2;

void Program1()
{
    while (turn2) { };
    turn1 = true;
    // critical section
    turn1 = false;
}
```

```
void Program2()
{
    while (turn1) { };
    turn2 = true;
    // critical section
    turn2 = false;
}
```

Пр2:

```
bool turn1, turn2;

void Program1()
{
    while (turn2) { };
    turn1 = true;
    // critical section
    turn1 = false;
}
```

```
void Program2()
{
    while (turn1) { };
    turn2 = true;
    // critical section
    turn2 = false;
}
```

- Изискването само един процес да е в критичната си секция е нарушено.

Пр3:

```
bool turn1, turn2;

void Program1()
{
    turn1 = true;
    while (turn2) { };
    // critical section
    turn1 = false;
}
```

```
void Program2()
{
    turn2 = true;
    while (turn1) { };
    // critical section
    turn2 = false;
}
```

Пр3:

```
bool turn1, turn2;

void Program1()
{
    turn1 = true;
    while (turn2) { };
    // critical section
    turn1 = false;
}
```

```
void Program2()
{
    turn2 = true;
    while (turn1) { };
    // critical section
    turn2 = false;
}
```

- Двата процеса могат да се блокират.

Пр4:

```
bool turn1, turn2;

void Program1()
{
    turn1 = true;
    while (turn2)
    {
        turn1 = false;
        // random sleep
        turn1 = true;
    };
    // critical section
    turn1 = false;
}
```

```
void Program2()
{
    turn2 = true;
    while (turn1)
    {
        turn2 = false;
        // random sleep
        turn2 = true;
    };
    // critical section
    turn2 = false;
}
```

Пр4:

```
bool turn1, turn2;

void Program1()
{
    turn1 = true;
    while (turn2)
    {
        turn1 = false;
        // random sleep
        turn1 = true;
    };
    // critical section
    turn1 = false;
}
```

```
void Program2()
{
    turn2 = true;
    while (turn1)
    {
        turn2 = false;
        // random sleep
        turn2 = true;
    };
    // critical section
    turn2 = false;
}
```

- Тук се избягва мъртвата хватка, но се стига до друг проблем "Безкрайно отлагане".

Синхронизация – машинно решение

- Машинна команда **T**estAnd**S**et

- TS(a, b) :

  1) a = b;

  2) b = true;

Пр:

```
bool common;
void Program1()
{
    var proc1 = true;
    while (proc1)
        TS(ref proc1, ref common);

    // critical section

    common = false;
}
```

```
void Program2()
{
    var proc2 = true;
    while (proc2)
        TS(ref proc2, ref common);

    // critical section

    common = false;
}
```

Пр:

```
bool common;
void Program1()
{
    var proc1 = true;
    while (proc1)
        TS(ref proc1, ref common);

    // critical section

    common = false;
}
```

```
void Program2()
{
    var proc2 = true;
    while (proc2)
        TS(ref proc2, ref common);

    // critical section

    common = false;
}
```

- Възможно е безкрайно отлагане, при наличие на приоритетно изпълнение.

Семафори

- Въведени от Дейкстра;
- Дефиниция:

  P(s): while(s == 0) { skip; } s = s-1;

  V(s): s = s + 1;

  Инициализация на s: s = const;

Семафори - употреба

```
Semaphore s;
void Program1()
{
    s.P();
    // critical section
    s.V();
}

void Program2()
{
    s.P();
    // critical section
    s.V();
}
```

Семафори – реализация с TS

```
class Semaphore
{
    bool s;

    public void P()
    {
        var wait = true;
        while (wait)
            TS(ref wait, ref s);
    }

    public void V()
    {
        s = false;
    }
}
```

Проблеми при синхронизацията

- Активно очакване (изразходват се ресурси в while);
- Безкрайно отлагане при приоритетно изпълнение.

Решение

- Вместо в активно очакване P(s) блокира процеса и го поставя в опашка (FIFO).

  V(s) вади процес от опашката и го активира.

Семафори

```
class Semaphore
{
    bool _status;
    Queue<Process> _queue = new Queue<Process>();

    public void P()
    {
        if (_status)
        {
            _status = false;
            return;
        }
        //процесът се блокира и се поставя в опашката
    }

    public void V()
    {
        if (_queue.Count() > 0)
        {
          //разблокиране на п-с от опашката
        }
        else
            _status = true;
    }
}
```

Събития

- Wait(s) - чака се настъпването на събитие;
- Signal(s) - сигнализира се за настъпване на събитие;
- Reset(s) - връща събитието в несигнализиран вид.

Събития

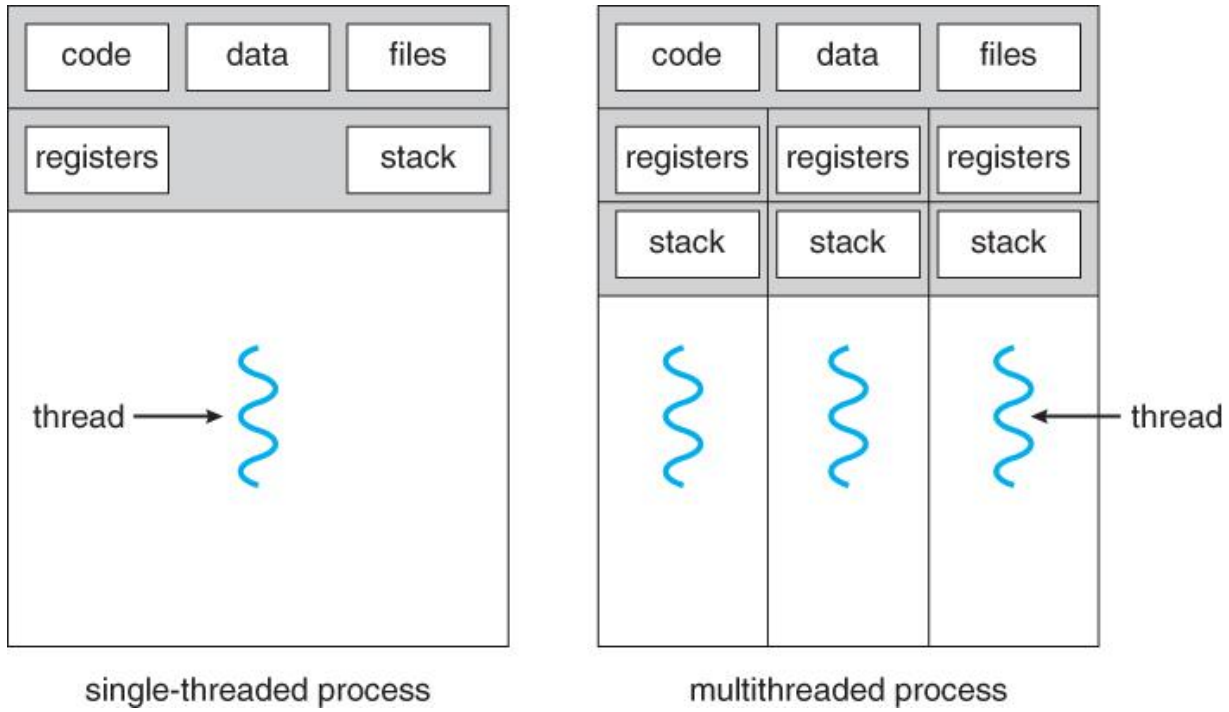
```
class Event
{
    bool _status;
    Queue<Process> _queue = new Queue<Process>();

    public void Wait()
    {
        if (_status)
            return;

        //процесът се блокира и се поставя в опашката
    }

    public void Signal()
    {
        _status = true;
        while (_queue.Count() > 0)
        {
            //разблокиране на п-с от опашката
        }
    }
}
```

Процеси и нишки



single-threaded process

multithreaded process

Процес

- Програма;
- Собствено адресно пространство;
- Отнема повече време за създаване, превключване, унищожение.

Нишка

- Функция;
- Общо адресно пространство;
- Отнема по-малко време за създаване, превключване, унищожение.

Процеси в C#

```
using System.Diagnostics;


                Process.Start("program.exe");

                Process process = new Process();
                process.StartInfo.FileName = "program.exe";
                process.StartInfo.Arguments = "-n";
                process.Start();

                process.WaitForExit();
                var status = process.ExitCode;
```

Комуникация между процеси

- Съобщения
  - Pipe;
  - Socket;
  - …
- Обща памет

Обекти за синхронизация в Windows

- Mutex;
- Semaphore;
- Event.

Нишки в C#

```
using System.Threading;

   public static void DoWork()
   {
   }

         Thread thread = new Thread(DoWork);
         thread.Start();

         var a = 5;
         var b = 6;
         var c = 0;
         Thread thread2 = new Thread(() =>
         {
             c = a + b;
         });
         thread2.Start();
```

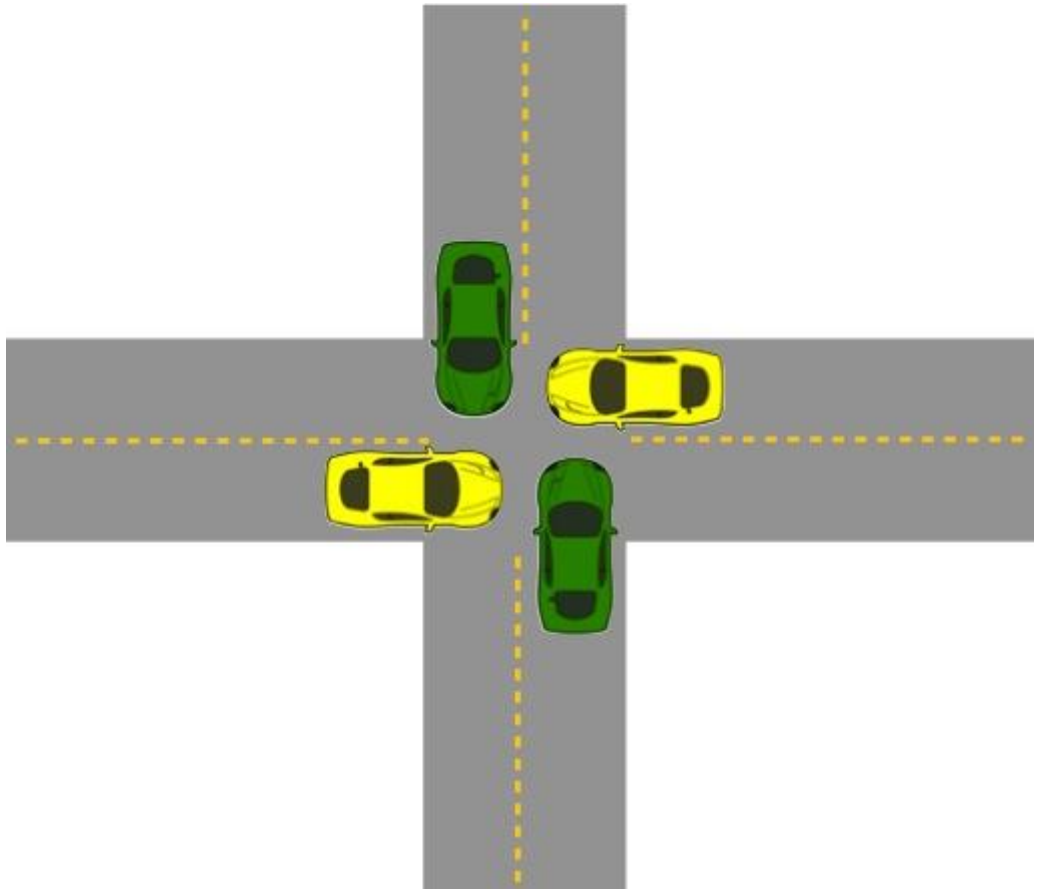
Комуникация между нишки

- Общо адресно пространство

Синхронизация в C#

- Езикови конструкции
  - Lock;
  - Await;
  - …
- Синхронизационни обекти
  - Mutex;
  - Semaphore;
  - Събития (AutoResetEvent, ManualResetEvent, …);
  - …

Мъртва хватка



Мъртва хватка - условия

- Взаимно изключване;
- Очакване на ресурси;
- Непреразпределение;
- Циклично очакване.

Мъртва хватка