# Процесс

- Процесс — экземпляр выполняемой программы, независимый объект, которому выделены системные ресурсы (например, процессорное время и память).
- Каждый процесс выполняется в отдельном независимом адресном пространстве
- Для взаимодействия процессов необходимо использовать специальные средства межпроцессного взаимодействия (IPC = inter process communication)
- Средства IPC: конвейеры, файлы, каналы (pipe), сокеты.
- Процесс является контейнером для потоков; каждый процесс состоит хотя бы из одного потока.

# Поток

Поток – единица выполнения на процессоре; сущность ОС, инкапсулирующая фрагмент программного кода для выполнения на процессоре

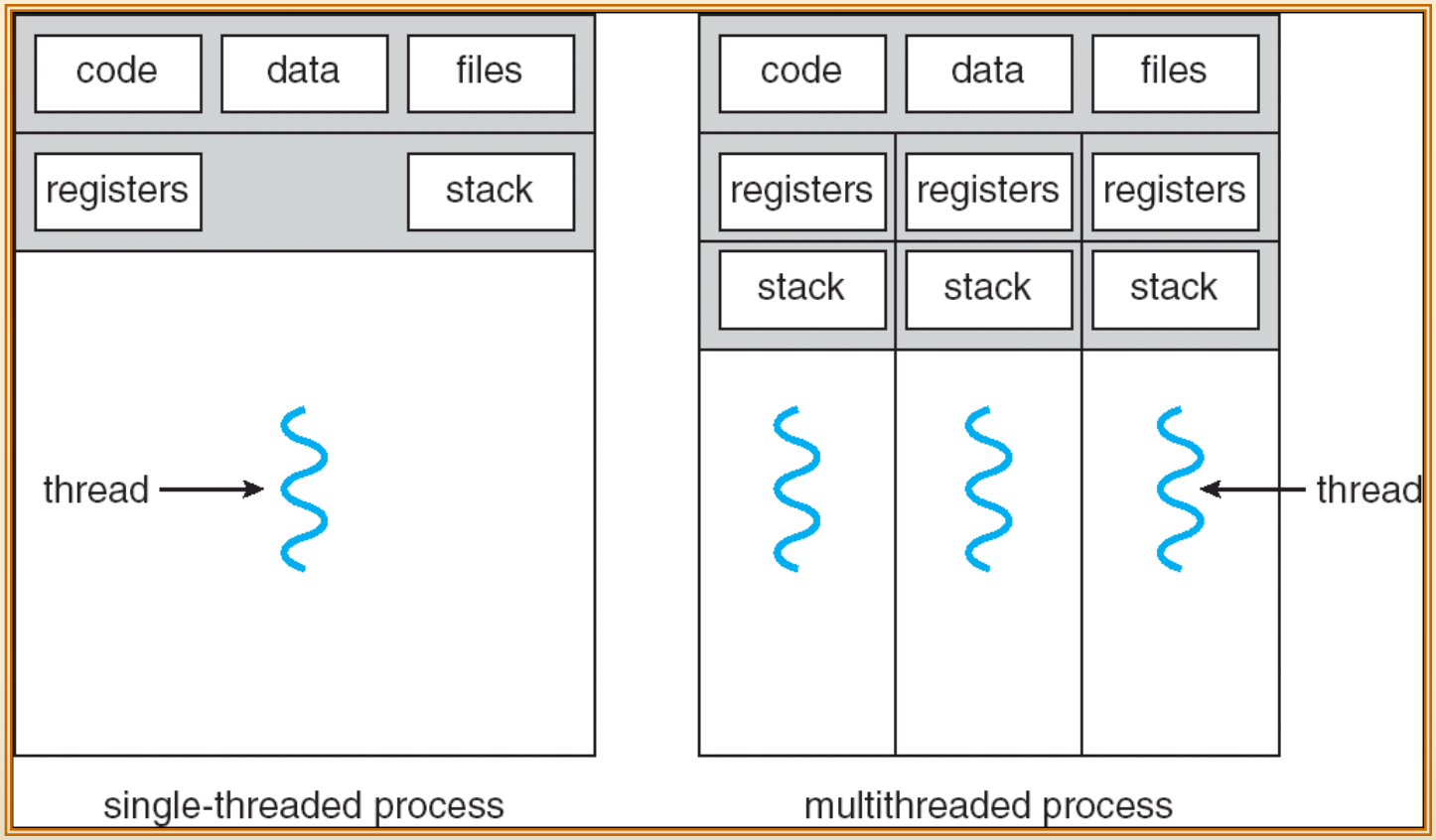
Поток состоит из:

- объекта ядра, через который операционная система управляет потоком и в котором хранится статистическая информация о потоке;
- стека потока, который содержит параметры всех функций и локальные переменные, необходимые потоку для выполнения кода.

Все потоки одного процесса разделяют:

- программный код;
- ресурсы: динамическая память,

# Многопоточность



# Потоки vs. Процессы

Многопоточность эффективнее:

- Меньшее время для переключения контекстов, так как не требуется перезагрузка виртуального адресного пространства
- Для взаимодействия не требуются IPC-средства
- Быстрота создания потока, по сравнению с процессом, примерно в 100 раз.

# «Легковесные» потоки

- Существует несколько реализаций «облегченных» потоков (cooperative threads): green threads, fibers, LWT, protothreads, user-level threads, task
- Требуют значительно меньше ресурсов, минимизируют количество системных вызовов
- Выполняются в контексте потока, который их создал и требуют лишь сохранения регистров процессора при их переключении.
- Управляются планировщиком уровня процесса; некоторые реализации fiber управляются планировщиком ОС
- Некоторые реализации легковесных потоков не поддерживают параллельность (элементы выполняются в рамках одного потока)

# Структура потока

- **Ядро потока (thread kernel)**. Структура содержит контекст потока, то есть блок памяти с набором регистров процессора (700 байт/1240/2500). Регистры содержат программный указатель и указатель стека. Программный счетчик содержит адрес инструкции, которую поток должен выполнить, а указатель стека ссылается на вершину стека потока.
- **Блок окружения потока.** Это место в памяти, выделенное и инициализированное в пользовательском режиме (адресное пространство, к которому имеет быстрый доступ код приложений). Блок занимает одну страницу памяти (4 Кб / 8 Кб) и содержит заголовок цепочки обработки исключений, локальное хранилище данных для потока и некоторые структуры данных, используемые интерфейсом графических устройств (GDI) и графикой OpenGL.
- **Стек пользовательского режима (user-mode stack).** Применяется для передаваемых в методы локальных переменных и аргументов. Также он содержит адрес, показывающий, откуда начнет исполнение поток после того, как текущий метод возвратит управление. По умолчанию на каждый стек пользовательского режима Windows выделяет 1 Мбайт памяти.
- **Стек режима ядра (kernel-mode stack).** Используется, когда код приложения передает аргументы в функцию операционной системы, находящуюся в режиме ядра. Ядро ОС вызывает собственные методы и использует стек режима ядра для передачи локальных аргументов, а также для сохранения локальных переменных

# Переключение контекста

ОС с вытесняющей многозадачностью

- Каждые 30 мс ОС осуществляет проверку готовых к выполнению потоков и передает управление потоку. При выборе потока учитываются приоритеты.

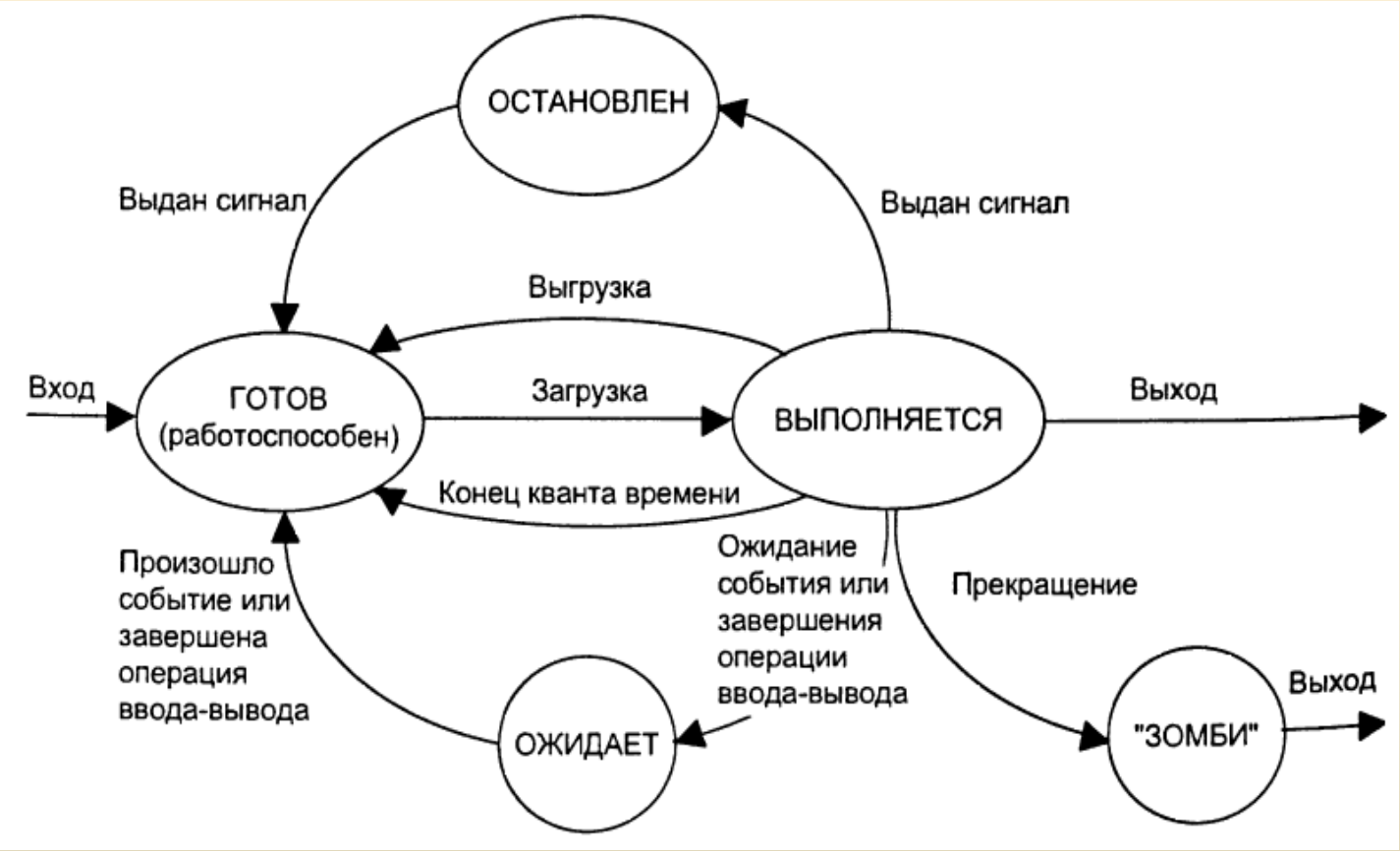
При переключении:

- Значения регистров процессора для исполняющегося в данный момент потока сохраняются в структуре контекста, которая располагается в ядре потока.
- Из набора имеющихся потоков выделяется тот, которому будет передано управление. Если выбранный поток принадлежит другому процессу, ОС переключает для процессора виртуальное адресное пространство.
- Значения из выбранной структуры контекста потока загружаются в регистры процессора.

# Состояние потока

- “Ready”: поток готов к выполнению; все готовые потоки помещаются в очереди готовности, в каждой такой очереди содержатся потоки с одинаковым приоритетом.
- “Running”: поток выбирается из очереди готовности, назначается процессору и переходит в состояние выполнения.
- Поток снимается с процессора, если его квант времени истек, или если перешел в состояние готовности поток с более высоким приоритетом. Выгруженный поток снова помещается в очередь готовности.
- “Waiting”: поток пребывает в состоянии ожидания, если он ожидает наступления некоторого события или завершения ввода-вывода.
- “Stoped”: поток прекращает выполнение, получив сигнал останова, и остается в этом состоянии до тех пор, пока не получит сигнал продолжить работу.

# Состояние потока



# Работа с потоками в C#

- Работа с потоками реализуется через объекты **Thread** пространства имен **System.Threading**
- Основные свойства потока:
  - **ManagedThreadId**
  - **ThreadState**
  - **ThreadPriority**
  - **IsAlive, IsBackground, IsThreadPoolThread**
  - **Name**
- К текущему потоку можно обратится через **System.Threading.Thread.CurrentThread**

# Работа с пользовательскими потоками

- При инициализации поток связывается с рабочей функцией (в полной записи через делегат)

  Полная форма:

  ThreadStart tWork = new ThreadStart(f);

  Thread thr1 = new Thread(tWork);

  Сокращенная форма:

  **Thread thr1 = new Thread(f);**

- В качестве рабочей функции можно использовать:
  - Метод объекта
  - Статический метод класса
  - Делегат
  - Анонимный делегат
  - Лямбда-выражение

```
class A
{     public void DoWork() { ..}      }
class Program
{
    static void OtherWork() { .. }
    static void Main()
    {
        A a = new A();
        // экземплярный метод
        Thread thread1 = new Thread(a.DoWork);
        // статический метод
        Thread thread2 = new Thread(OtherWork);
        // анонимный метод
        Thread thread3 = new Thread(() => { .. });
    }
}
```

# fork/join

- Запуск потока осуществляется с помощью метода **Start**:

```
Thread thr = new Thread(..);
thr.Start();
```

- Блокировка текущего потока для ожидания завершения созданного осуществляется с помощью метода **Join**:

```
Thread thr = new Thread(..);
thr.Start();
thr.Join();
```

# Передача параметров

- Допускаются две сигнатуры для рабочей функции:

```
void ThreadFunction();
void ThreadFunction(object o);
```

- Передача параметра осуществляется при запуске потока:

```
Thread thr = new Thread(Calc);
thr.Start(1000);
```

- Передача произвольного числа параметров

```
static void Main()
{
    object[] ManyParams =
            new object[] {SomeObj, “Thread1”, 3.14};
    var thr = new Thread(DoWork);
    thr.Start(ManyParams as object);
}
static void DoWork(object o)
{
    SomeClass SomeObj = ((object[])o)[0] as SomeClass;
    string Name = (string)((object[])o)[1];
    double d = (double)((object[]o)[2];
```

- Использование параметров в лямбда-выражении

```
static void Main()
{
    string Name = “Thread 1”;
    int n = 100;
    double[] arr = ..;
    Thread thr = new Thread(() => {
            Console.WriteLine(Name);
            f(arr, n);
    });
    thr.Start();
}
```

# Захват переменных в лябда-выражениях

```
for(int i=0; i<10; i++)
{
    Thread t = new Thread(() =>
                    Console.Write(“ABCDEFGHIJK”[i]));
    t.Start();
}
```

# Приостановление потока

- Метод **Sleep** позволяет приостановить выполнение текущего потока на заданное число миллисекунд, при этом:

  - поток отказывается от остатка выделенного кванта времени;

  - если в качестве аргумента указывается ноль, то выполняющийся поток отдает выделенный квант времени и без ожидания включается в конкуренцию за процессорное время.

- Метод **Yield** также возвращает выделенный квант времени выполняющегося потока, но только в том случае, если есть конкуренция за процессор (или процессорное ядро). Управление может быть передано и потоку с меньшим приоритетом.

# Привязка потоков

- Большинство ОС использует *нежесткую привязку* (soft affinity) потоков к процессорам. Это означает, что при прочих равных условиях, система пытается выполнять поток на том же процессоре, на котором он работал в последний раз.
- Применение жесткой привязки позволяет указать процессоры, которые могут использоваться для выполнения потоков процесса:

  ProcessorAffinity = (IntPtr) 1
- Указание предпочтительного процессора:

  IdealProcessor = 0;

# Приоритеты потоков

- Приоритеты потоков определяют очередность выделения доступа к ЦП. Высокоприоритетные потоки имеют преимущество и чаще получают доступ к ЦП, чем низкоприоритетные.
- На уровне ОС приоритет потока находится в диапазоне от 0 (самый низкий) до 31 (самый высокий).
- Windows поддерживает 6 классов приоритета для процесса и 7 уровней относительного приоритета потока
- Класс приоритета:

  Process.GetCurrentProcess().**PriorityClass**
- Относительный приоритет:

  Thread.CurrentThread.**Priority**

# Приоритеты потоков

| Относительный приоритет потока | Idle | Класс приоритета процесса | | | | Real-time |
|---|---|---|---|---|---|---|
| | | Below normal | Normal | Above normal | High | |
| Time-critical (критичный по времени) | 15 | 15 | 15 | 15 | 15 | 31 |
| Highest (высший) | 6 | 8 | 10 | 12 | 15 | 26 |
| Above normal (выше обычного) | 5 | 7 | 9 | 11 | 14 | 25 |
| Normal (обычный) | 4 | 6 | 8 | 10 | 13 | 24 |
| Below normal (ниже обычного) | 3 | 5 | 7 | 9 | 12 | 23 |
| Lowest (низший) | 2 | 4 | 6 | 8 | 11 | 22 |
| Idle (простаивающий) | 1 | 1 | 1 | 1 | 1 | 16 |

# Динамический приоритет

- Для потока определены свойства BasePriority, CurrentPriority
- Приоритет потока может временно повышаться системой по сравнению с базовым приоритетом в пределах 1 – 15. Причины:

  - драйверы устройств (клавиатура);

  - голодание потока в течении продолжительного времени (3 – 4 с);

  - активное приложение;
- Свойства PriorityBoostEnabled для процесса и потока позволяют отключать режим возможного динамического изменения приоритета.

# Доступность переменных

- Переменные, объявленные внутри рабочей функции или метода, вызванного из рабочей функции, являются *приватными* или *локальными*

```
public void f() { int x = 3; }
```

- Переменные, объявленные в классе, содержащем рабочую функцию, по умолчанию, являются разделяемыми (shared)

```
class Work {
        int x;
        public void f() { x = 3; }
}
```

# Доступность переменных

| | Внутри метода | Вне метода |
|---|---|---|
| **Статический класс** | **local** | **shared** |
| **Единственный объект** | **local** | **shared** |
| **Разные объекты** | **local** | **local** |

# Приватные данные для потоков

Для введения приватных элементов класса можно использовать:

- Атрибут ThreadStatic для статических полей
- Объект LocalThread<T>
- Именованные области локального хранилища потоков

# Пул потоков

- Используется для автоматизации распределения пользовательских рабочих элементов по рабочим потокам
- Добавление рабочего элемента в пул:

  ThreadPool.QueueUserWorkItem(f, o), где f – метод, o – параметр типа object;
- В качестве рабочего элемента могут быть: методы, делегаты, лямбда-выражения