# Операционные системы

Курс лекций для гр. 4057/2

**Лекция №6**

# Содержание

## Раздел 2. Взаимодействие процессов

2.1 Синхронизация
2.2 Тупики

} Лекции 4-5

2.3 Коммуникация
2.4 Средства многопоточного программирования

# Понятие коммуникации взаимодействующих процессов

Коммуникация – это обмен данными; он возможен:

- Через общую область памяти:
  - у потоков одного и того же процесса это - общее адресное пространство
  - для разных процессов объявляется совместно используемая область памяти

Это невозможно для процессов на разных машинах в распределенной системе

- Путем передачи сообщений; плюсы этого способа:
  - работает в распределенной системе
  - упорядочение обмениваемых данных во времени
  - сигнализация о посылке и получении данных
  - в системе с общей памятью обеспечивает более надежный (но медленный) обмен – см. Windows: message-driven

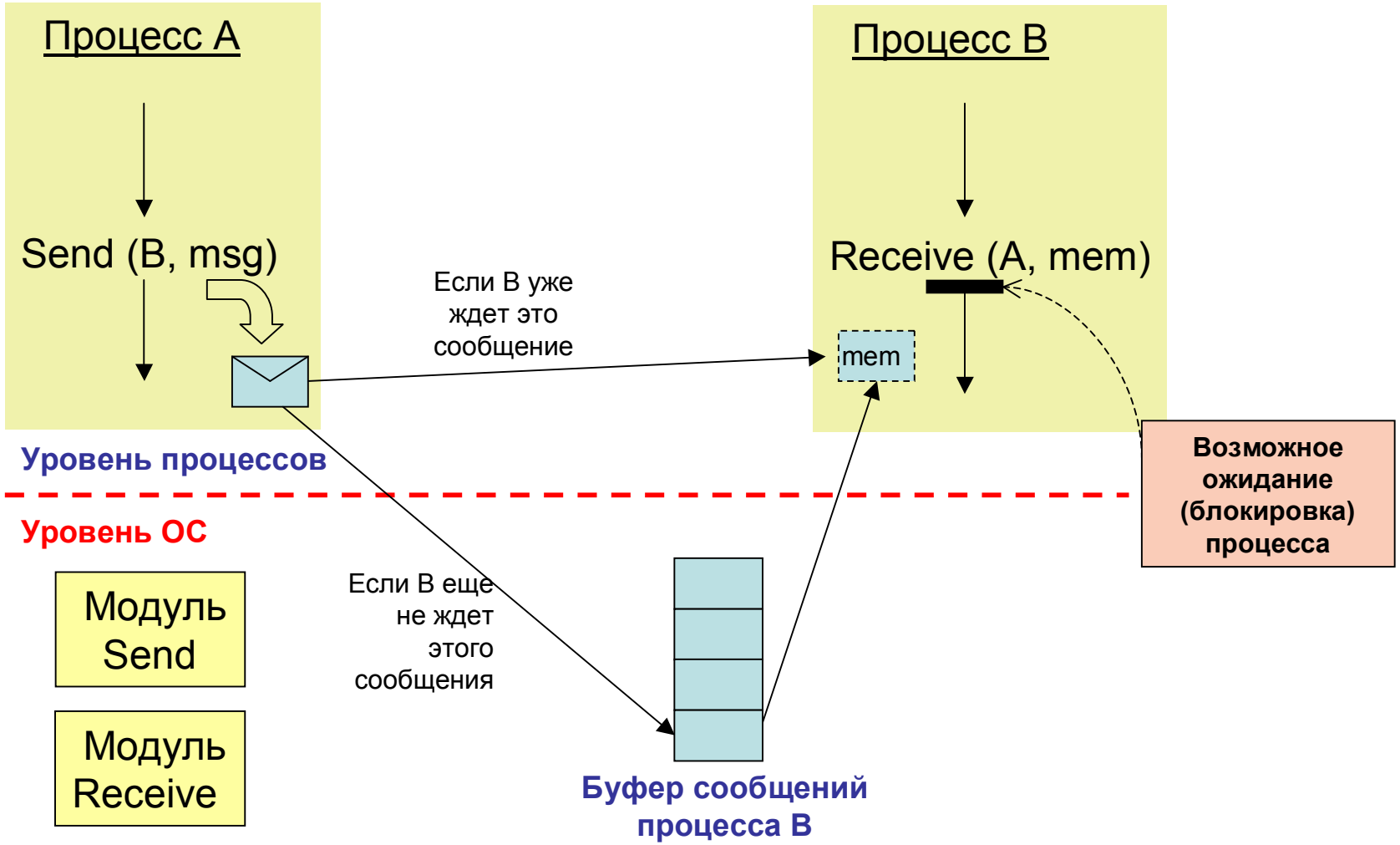
Далее под коммуникацией понимаем обмен данными путем передачи сообщений

# Взаимосвязь синхронизации и коммуникации

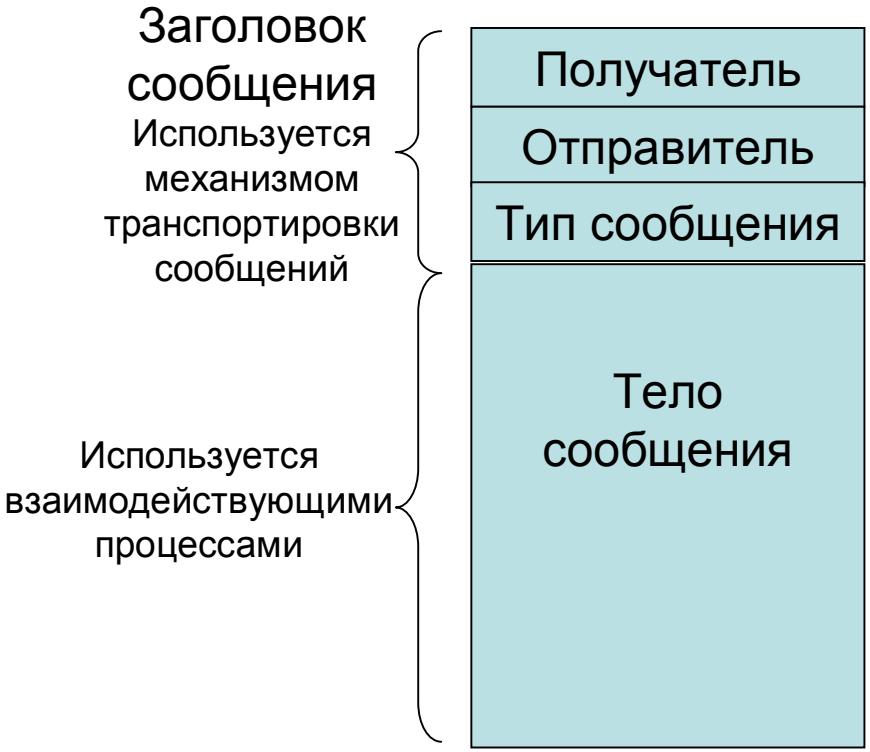
- Их различие  относительно: это условное различие между сигналами и сообщениями
  - сигналы синхронизации можно считать сообщениями (короткими: 1 бит)
  - сообщения можно трактовать как сигналы, предполагающие определенную реакцию, в частности, как сигналы синхронизации
- Передача сообщений требует синхронизации процессов отправителя и приемника

Взаимозаменяемость: передачу сообщений можно использовать с целью синхронизации и наоборот, строить схемы обмена сообщениями с помощью синхронизационных примитивов

# Типичная схема механизма передачи сообщений в ОС



# Структура сообщения




**Тело сообщения состоит из полей, содержащих типизированные данные – вроде тех, что при вызове процедуры передаются в качестве аргументов**

Совокупность средств передачи сообщений определенного типа назыв. логическим каналом связи

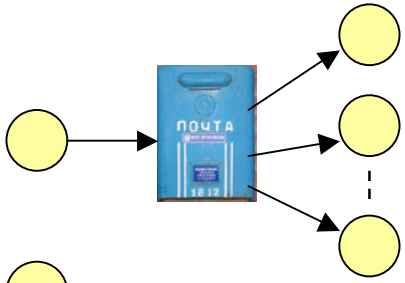
# Варианты логических каналов связи

- Непосредственная (direct) коммуникация, или связь с прямой адресацией:
  - явная (симметричная) адресация: связь устанавливается строго между двумя процессами: send(B, msg), receive (A, mem)
  - неявная (асимметричная) адресация: receive (proc_id, msg) - получить сообщение от *любого* процесса, где proc_id - выходная переменная; в нее возвращается имя процесса-отправителя

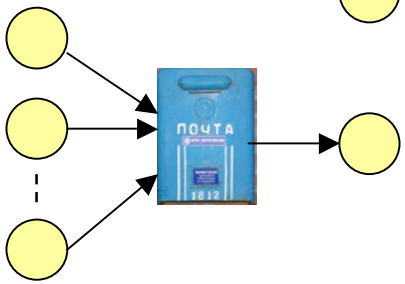
  Недостаток схемы - ограниченная модульность программ процессов: изменение имени любого процесса потребует изменения ссылок на него в других процессах и перекомпиляции соответствующих программ
- Непрямая (indirect) коммуникация, или связь с косвенной адресацией: сообщения посылаются (кладутся) в *почтовые ящики* (или *порты*) и получаются (вынимаются) из них
  - Почтовый ящик – это циклический буфер для сообщений определенного типа; тип задается при его создании
  - ОС обеспечивает две функции: **send**(mbox_name, msg) и **receive**(mbox_name, msg).

# Отношения между отправителем и получателем при непрямой связи

«Один к одному»: закрытая связь между двумя процессами



«Один ко многим»: такая связь называется широковещательной (broadcasting)



«Многие к одному»: такой почтовый ящик часто называется портом

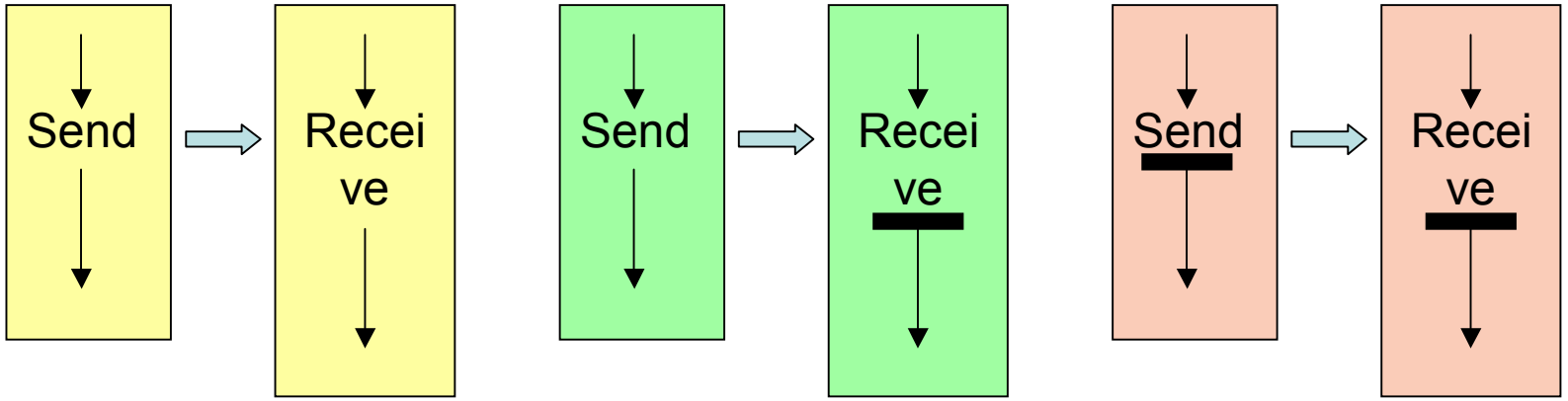
«Многие ко многим»: глобальный порт

# Служба почтовых ящиков в ОС

- Владелец почтового ящика – процесс
  - Процесс-владелец объявляет переменную типа mailbox; он может только получать сообщения из этого ящика
  - Другие процессы - пользователи ящика – могут только отправлять их
  - При уничтожении процесса-владельца ящик автоматически уничтожается
- Владелец почтового ящика - ОС
  - ОС может разрешать процессам создавать и уничтожать их явно
  - Связь процесса с ящиком может быть статической или динамической; для порта и «один к одному» она обычно статическая

# Синхронная и асинхронная передача сообщений

| Отправление | Получение | |
|---|---|---|
| | Неблокирующее | Блокирующее |
| Неблокирующее | *Полностью асин-хронная схема* | *Асинхронная схема* |
| Блокирующее | *Не применяется* | *Рандеву* |




Используется для некритичных сигналов

Наиболее часто используемая схема

Рандеву - полностью синхронная схема (Ada)

# Синхронизация получателя

- Минус неблокирующего получения: возможность неполучения сообщений, отправленных после того, как получатель выполнил receive
- Минус блокирующего получения – опасность бесконечного ожидания получателем, если процесс-отправитель завершился или завис
  - Предотвращение этого: таймирование - параметр timeout в операции receive, ограничивающий время ожидания

# Синхронизация отправителя

- Недостатки неблокирующего отправления – опасность зацикленной генерации сообщений при некоторой ошибке и отсутствие отслеживания успешной доставки сообщения

  Варианты такого отслеживания:

  – ожидание отправителем подтверждения приема

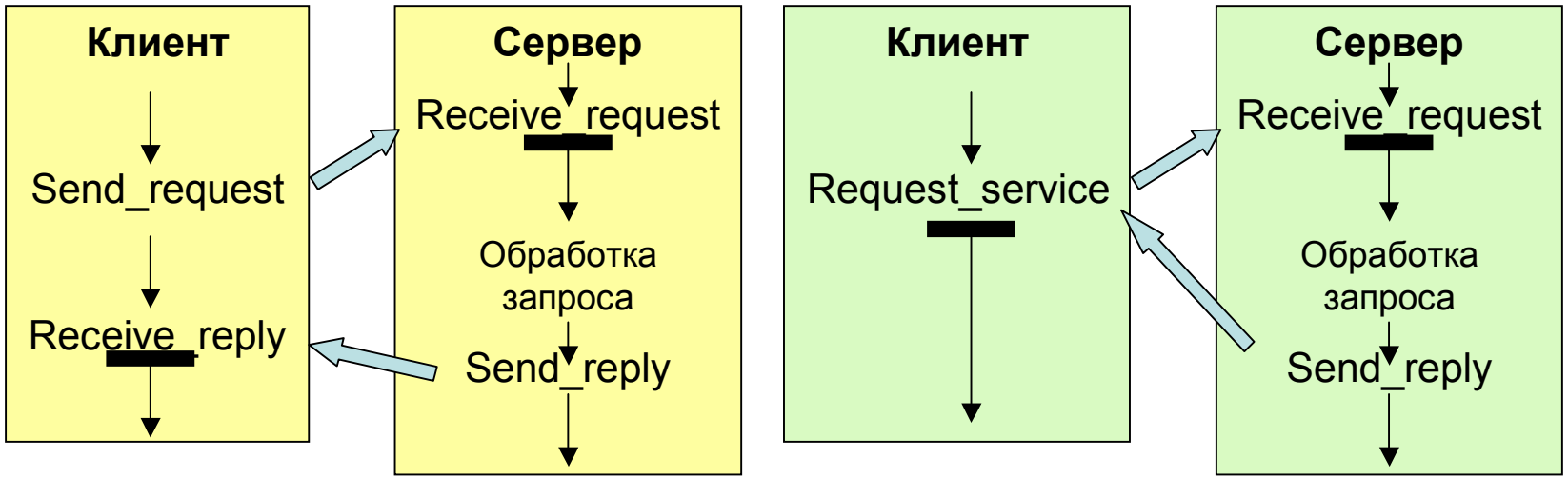
| Отправитель P | | Получатель Q |
|---|---|---|
| send(Q, msg) | ⇒ | receive(P, msg) |
| receive(Q, msg) | ⇐ | send(P, acknowgmt) |

  – Блокирование (возможно, таймированное) отправителя P непосредственно в операции send, пока не получено подтверждение от получателя, выполнившего операцию reply(P, acknowgmt) – т.е., точный аналог рандеву

- Общие недостатки синхронной передачи сообщений по сравнению с асинхронной
  – Снижается степень параллелизма из-за вынужденного ожидания отправителя
  – Возрастает опасность ошибок, приводящих к тупикам: напр., два процесса ожидают на send выполнения receive друг у друга

# Коммуникация клиент-север

Сервер – это процесс, регулярно обрабатывающий *запросы* от процессов-клиентов и возвращающий им *ответы*

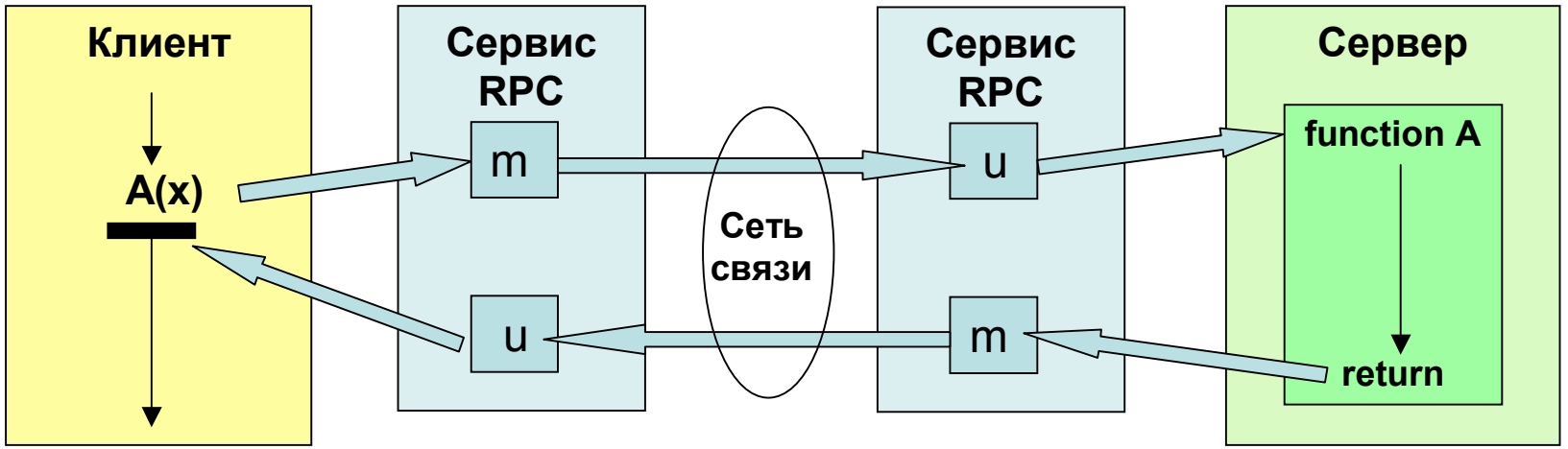



Асинхронная коммуникация

Синхронная коммуникация - совмещение запроса с ожиданием ответа – аналог вызова процедуры: *удаленный вызов процедур – Remote Procedure Call (RPC)*

# Удаленный вызов процедур

Вызов удаленной процедуры для программиста не отличается от вызова локальной процедуры. Сервис RPC и сетевые протоколы преобразуют вызов в обмен сообщениями.




- Имена процедур привязываются к сетевым адресам; есть служба поиска
- m – marshalling – упаковка аргументов вызова x и результатов работы серверной подпрограммы **function A** в пакет передачи данных
- u – unmarshalling - распаковка

# Средства коммуникации в Unix/Linux

1. Сигналы – однобитовые сообщения, аналог прерываний; обрабатываются получателем немедленно или сразу после перехода в состояние выполнения. Примеры:

   SIGKILL – завершить процесс

   SIG CLD – завершился сыновний процесс

Сигналы имеют буфер размером в одну порцию, поэтому сообщение может быть потеряно: если другой сигнал того же типа послан до того, как предыдущий сигнал воспринят процессом-адресатом, то первый перекрывается вторым.

Процесс – получатель может реагировать на сигнал одним из трех способов:

- игнорировать его
- реагировать по умолчанию
- обрабатывать в запрограммированном обработчике (handler)

2. Сообщения – блоки текста определенного типа. С каждым процессом связана FIFO очередь сообщений – аналог почтового ящика

- В системном вызове `msgsnd` указывается тип сообщения; отправитель блокируется при заполненной очереди
- Вызов `msgrcv`:
  - с указанием типа: если сообщений такого типа нет, получат. не блокируется
  - без указания типа сообщения: получатель блокируется, если очередь пуста

# Средства коммуникации в Unix/Linux (2)

3. Каналы (pipes) – циклические буферы по схеме производитель-потребитель с порциями в 1 байт для связи двух процессов друг с другом

- Каналы удобны для программирования цепочки «конвейерных» вычислений
- Канал является наследуемым ресурсом и имеет пару структур данных типа wait.queue и signal.queue для синхронизации отправителя и получателя

4. Сокеты (sockets) – аналоги каналов, только двунаправленные

- Сокеты обычно связывают процессы на разных машинах в распределенных системах

Два вида сокетов:

- Потоковый (stream) – передает поток байтов
- Дейтаграммный (datagram) – передает порции данных (дейтаграммы)

# Средства коммуникации в Windows

1. Каналы аналогичны каналам Unix с такими отличиями:
   - порциями могут быть 128-байтовые сообщения
   - именованные каналы могут работать через сеть
2. Почтовые ящики - подобны каналам, но:
   - могут связывать не только два процесса - работать со многими отправителями и/или получателями
   - могут связывать удаленные процессы (без гарантии доставки)

   Отправитель:
   - неблокирующие операции Win32 `PostMessage`, `PostThreadMessage`
   - блокирующие `SendMessage`, `SendThreadMessage` или `SendMessageCallback`

   Получатель: `GetMessage` - всегда блокирующая операция
3. Сокеты - как в Unix

# Средства коммуникации в Windows (2)

4. Удаленный вызов процедур (RPC) и локальный вызов процедур (LPC) – универсальный способ связи клиент/сервер

- LPC – упрощенный RPC для связи клиента с сервером на одной машине, без сети, через память:
  - недокументированный, недоступный непосредственно пользователям
  - RPC в приложениях автоматически преобразуется в LPC

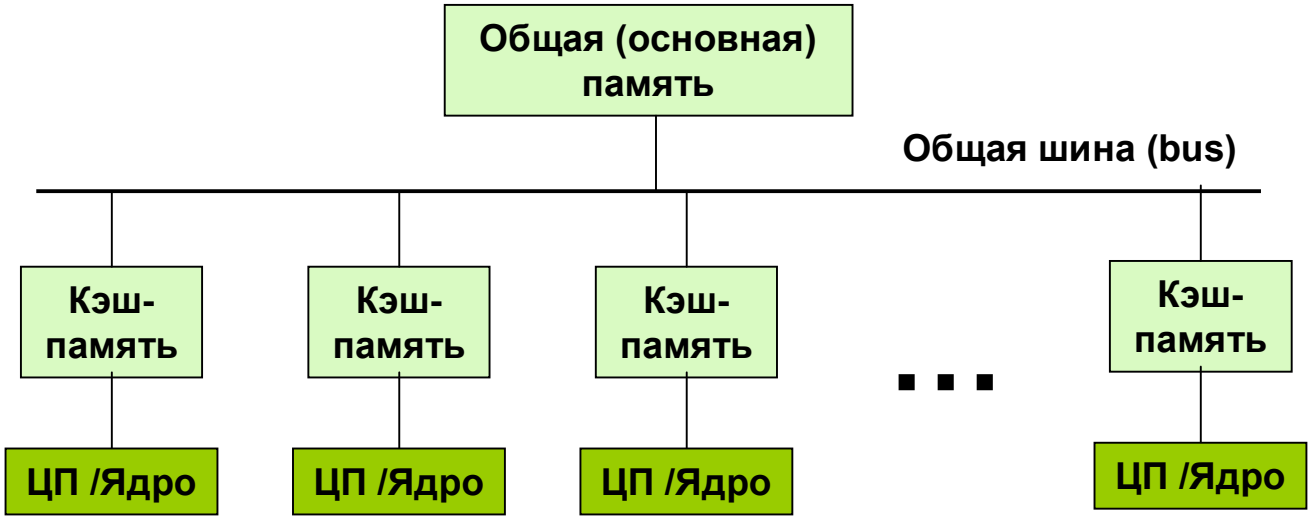
• Windows поддерживает разнообразные подсистемы (сервисы), с которыми приложения связываются через передачу сообщений

• Приложения рассматриваются как клиенты сервера подсистемы; сервер обычно запускает отдельный поток для обслуживания каждого клиента

• Этот способ – менее эффективный, чем обмен через общую память, поэтому Windows проигрывает Unix/Linux по реактивности

Это задумано для лучшей модульности, для сокращения времени реализации новых функциональностей и унификации взаимодействия в централизованных и распределенных системах

# Виды многопроцессорных структур




Многоядерные структуры = многопроцес-сорным

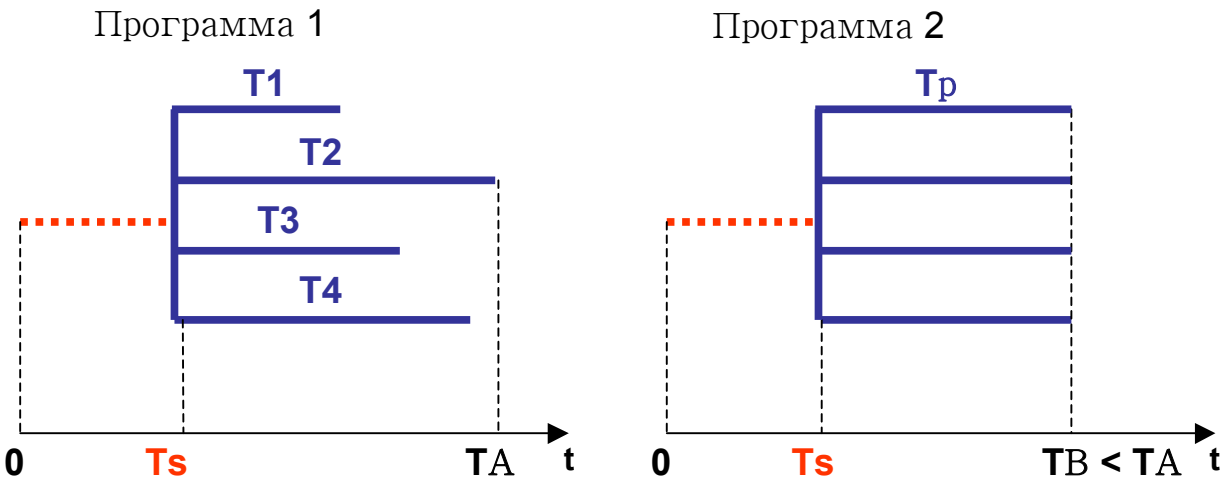
- Симметричная мультипроцессорность (SMP): одинаковые процессорные элементы с одинаковым доступом к общей памяти
  - ✓ Хорошо соответствует модели процесс-потоки процесса
  - ✓ Наиболее распространенная
- Асимметричная мультипроцессорность (ASMP): отдельные специализированные процессоры используются для различных задач
  - ✓ Это в видеоускорителях, игровых приставках и др. специализированных машинах
- Неоднородная структура памяти (Non-Uniform Memory Access, или NUMA): каждый процесс имеет приоритетный доступ к своей части общей памяти
  - ✓ В больших серверах

# Проблемы многопоточного программирования

- Тривиальная задача: независимые процессы = независимым приложениям
  - Необходимо только взаимное исключение работы с системными критическими ресурсами и борьба с тупиками
  - Повышает производительность системы, но не ускоряет выполнение отдельной программы
  - Что, если в ЦП 100 ядер?
- Нетривиальная задача: распараллеливание программы на взаимодействующие потоки для ее ускорения
  - Параллельные алгоритмы
  - Балансировка загрузки ядер или процессоров
  - Синхронизация и коммуникация

  PS: процессы офисных приложений были разбиты на потоки не из соображений их ускорения, а ради улучшения структурности и реактивности

# Закон Амдала




Последовательные операции алгоритма, выполняемые на одном процессоре:

Параллельные ветви алгоритма выполняемые на разных процессорах:

Несбалансированная нагрузка 4-х процессоров

Идеально сбалансированная нагрузка: **Tp = (T1 + T2 + T3 + T4) / 4**

Теоретический предел выигрыша производительности при физическом SMP-параллелизме определяется законом Амдала (Amdahl, 1967):

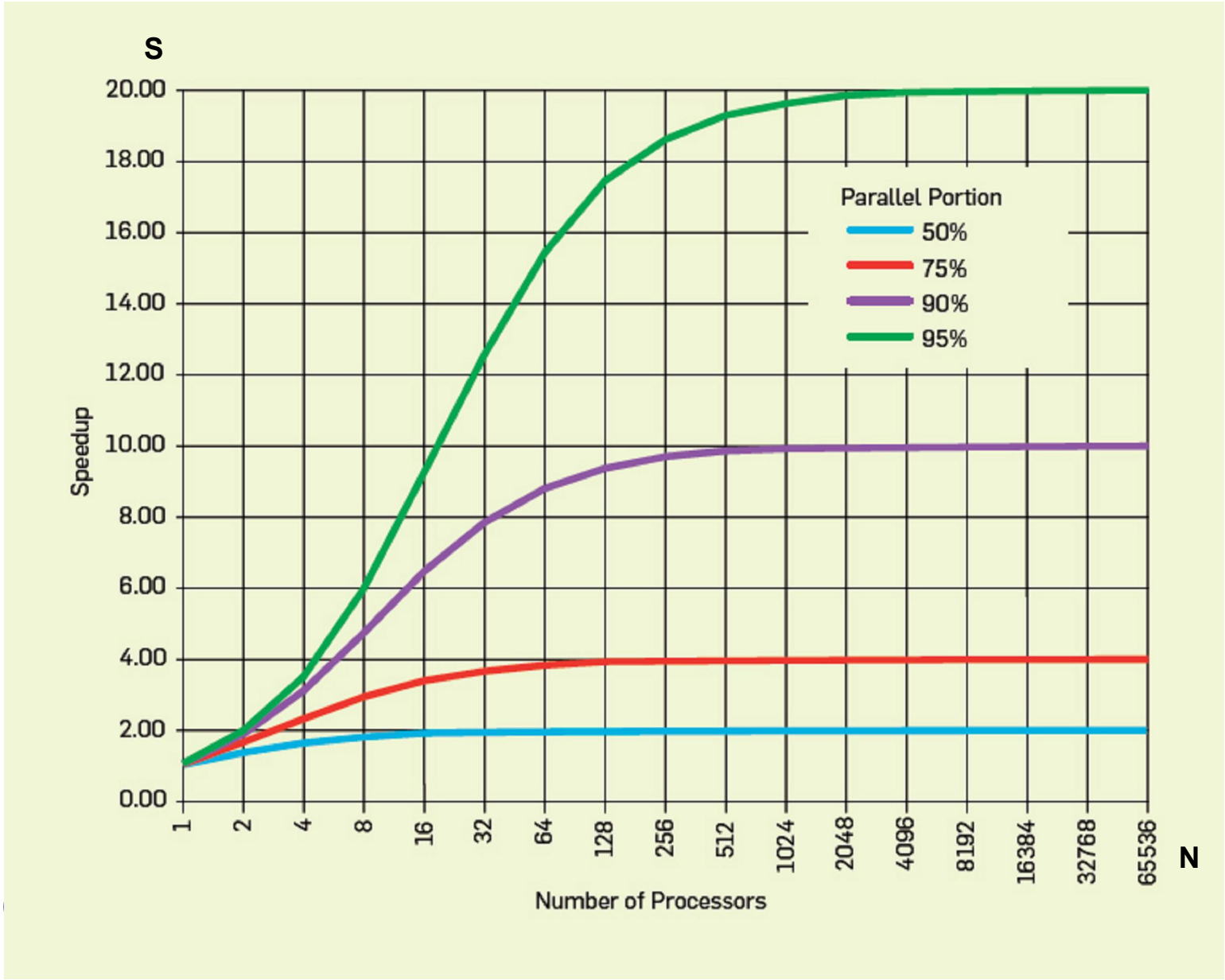
**S ≤ 1/(c + (1-c)/N),** где

**S** – коэффициент ускорения по сравнению с однопроцессорным вариантом

**c** - доля кода программы, выполняемого последовательно: **c = Ts / (Ts+T1+T2 +T3+T4)**

**N** - число процессоров

# График закона Амдала



# Следствия закона Амдала

- Коэфф. ускорения S не может быть больше 1/c при любом N
  - Напр., при c=0,25  $S \le 4$
- При c > 0 (что всегда) ускорение S не пропорционально числу процессоров и тем меньше, чем больше c
  - Напр., при c=0,05 и N=32 ускорение $S \le 12,5$
- Предельные оценки S еще ниже, если учесть простои процессоров из-за конфликтов обращения к общей памяти
  - Доступ к памяти – узкое место; рост ее скорости и пропускной способности шины – медленнее закона Мура
- Меры по увеличению S:
  - по возможности минимизировать c – долю фрагментов кода, которые выполняются последовательно: ввод-вывод, инициализация переменных, критические области, фрагменты, зависящие по данным, и пр.
  - обеспечивать максимально сбалансированную нагрузку процессоров – в идеале одинаковую
  - хранить данные во время обработки в локальной памяти, чтобы уменьшить интенсивность обращений к общей памяти

# Способы создания параллельных программ

1. Библиотеки управления потоками ОС
   - Win32 API
   - POSIX (Unix/Linux)
2. Библиотека OpenMP
   – оболочка над 1.
3. Библиотеки передачи сообщений
   - MPI
4. Автоматическое распараллеливание
   - в зачаточном состоянии

# Библиотека OpenMP

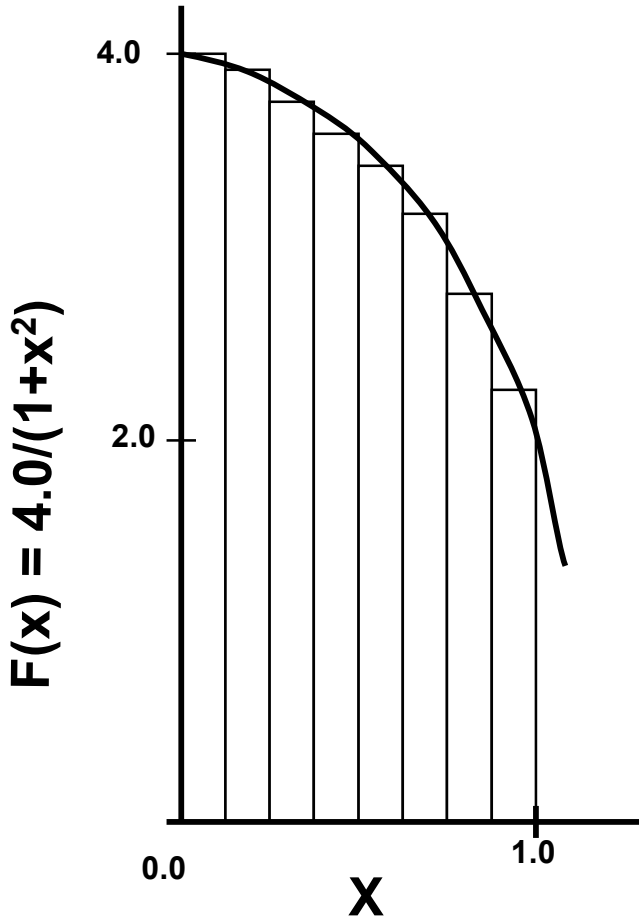
- OpenMP дает простой способ создания многопоточных приложений: не нужно думать о создании, синхронизации и уничтожении потоков
  - заботой программиста остается сам параллельный алгоритм
- Стандарт OpenMP разработан в 1997 г. первоначально для Фортрана
- Сейчас он включает в себя и C/C++; OpenMP поддерживают Visual C++ (начиная с 2005), компилятор C++ Intel, платформа Xbox 360 и др.
- Информация о распараллеливании алгоритма передается посредством директив компилятора – *прагм,* напр.:

```
#pragma omp parallel
```

- Можно установить режим игнорирования прагм, и тогда генерируется однопоточный (последовательный) код – это удобно для отладки
- Функции библиотеки периода выполнения служат для изменения и получения параметров среды и для поддержки синхронизации потоков

# Пример: вычисление числа $\pi$

(из курса В.Крюкова и В. Бахтина «Параллельное программирование с OpenMP» , http://www.intuit.ru/department/supercomputing/paralprogomp/)




$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

Мы можем аппроксимировать интеграл как сумму прямоугольников:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Где каждый прямоугольник имеет ширину $\Delta x$ и высоту $F(x_i)$ в середине интервала

# Вычисление числа $\pi$: последовательная программа

```
#include <stdio.h>
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = 1; i <= n; i ++)
    {
                x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
     }
     pi = h * sum;
     printf("pi is approximately %.16f”, pi);
     return 0;
}
```

# Программа с Win32 API (1/2)

```
#include <stdio.h>
#include <windows.h>
#define NUM_THREADS 2
CRITICAL_SECTION hCriticalSection;
double pi = 0.0;
int n =100000;
void main ()
{
   int i, threadArg[NUM_THREADS];
   DWORD threadID;
   HANDLE threadHandles[NUM_THREADS];
   for(i=0; i<NUM_THREADS; i++) threadArg[i] = i+1;
   InitializeCriticalSection(&hCriticalSection);
   for (i=0; i<NUM_THREADS; i++)  threadHandles[i] =
       CreateThread(0,0,(LPTHREAD_START_ROUTINE)
   Pi,&threadArg[i], 0, &threadID);
   WaitForMultipleObjects(NUM_THREADS, threadHandles,
   TRUE,INFINITE);
   printf("pi is approximately %.16f", pi);
}
```

# Программа с Win32 API (2/2)

```
void Pi (void *arg)
{
   int i, start;
   double h, sum, x;
   h   = 1.0 / (double) n;
   sum = 0.0;
   start = *(int *) arg;
   for (i=start; i<= n; i=i+NUM_THREADS)
   {
       x = h * ((double)i - 0.5);
       sum += (4.0 / (1.0 + x*x));
    }

  EnterCriticalSection(&hCriticalSection);
       pi += h * sum;
  LeaveCriticalSection(&hCriticalSection);
}
```

# Программа с Open MP

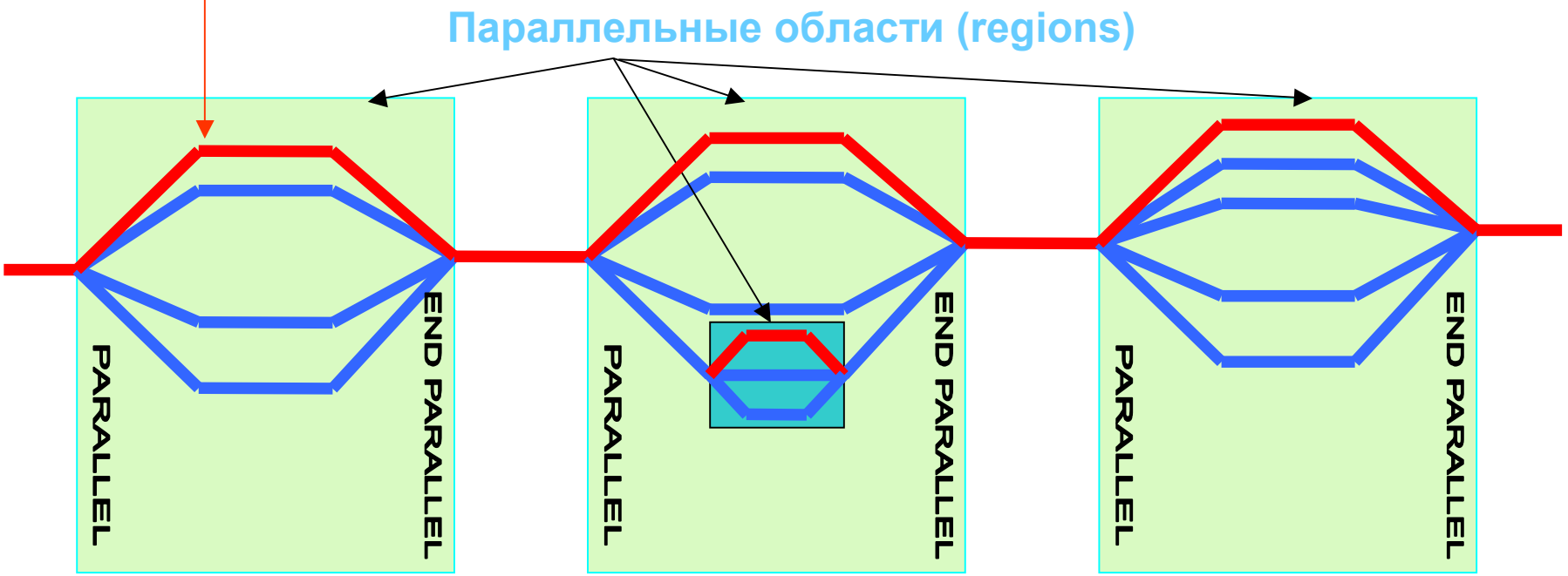
```
#include <stdio.h>
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h   = 1.0 / (double) n;
    sum = 0.0;
#pragma omp parallel for reduction(+:sum)
   private(x) // локальная копия данных
    for (i = 1; i <= n; i ++)
    {
       x = h * ((double)i - 0.5);
       sum += (4.0 / (1.0 + x*x));
    }
    pi = h * sum;
    printf("pi is approximately %.16f”,pi);
    return 0;
}
```

Тело цикла выполняется разными потоками для разных значений i параллельно

# Выполнение OpenMP-программы

Fork-Join параллелизм:

- Главный поток порождает группу потоков по мере необходимости
- Параллелизм добавляется инкрементально
- Число потоков в группе задается переменной окружения или может изменяться динамически перед началом выполнения параллельной области



# Основной прием – распараллеливание независимых циклов

- Для программы на слайде 30 по умолчанию будет создано N потоков в параллельной области (N = числу физических процессоров); каждый выполнит 1 /N общего числа итераций цикла
  - Напр., при N=4: 25000 итераций; первый поток - для i от 1 до 25000, второй для i от 25001 до 50000, и т.д.)
- Число N можно узнать: функция omp_get_num_threads и установить: omp_set_num_threads
- Поток может узнать свой № в группе: omp_get_num_proc и число процессоров в машине: omp_get_thread_num
- Алгоритмы численных методов обычно содержат много длительных циклов и поэтому хорошо соответствуют модели OpenMP

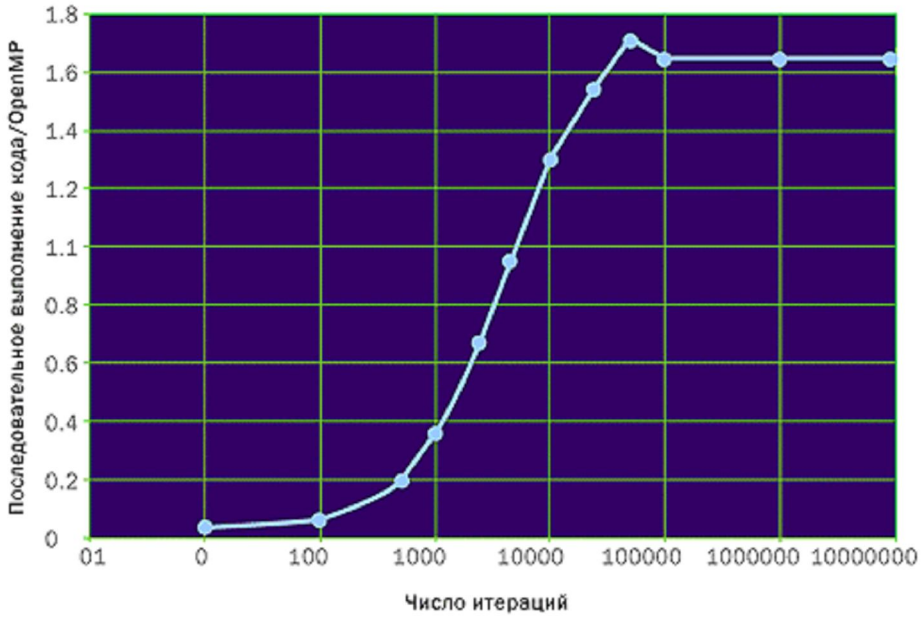
# Некоторые средства синхронизации в OpenMP

- #pragma omp critical: следующий блок - критическая секция
- #pragma omp single: следующий блок должен быть выполнен только одним потоком (любым)
- #pragma omp barrier: барьерная синхронизация; пока все потоки не достигнут этой точки, ни один поток не сможет продолжить выполнение
- omp_set_lock: запереть замок
- omp_unset_lock: открыть замок

Эти директивы и функции реализованы с помощью примитивов синхронизации ОС в Windows, Unix и др. ОС

# Эффективность распараллеливания

- Порождение параллельных потоков ведет к издержкам времени – это недостаток модели Fork=Join
- Выигрыш от параллельного выполнения циклов тем больше, чем больше трудоемкость тела цикла
- Для очень простого цикла на слайде и при двух процессорах параллельная версия обгоняет последовательную по быстродействию только примерно после 5000 итераций:



# Заключение

- Средства обмена сообщениями в ОС обычно используются для организации информационного взаимодействия процессов, но не потоков
  - Потокам достаточно обмена данными через общую память
- Наиболее популярные средства в Windows – RPC и почтовые ящики, а в Unix/Linux – сигналы и каналы
- Наиболее распространенная многопроцессорная структура – симметричная (SMP)
- Основная цель OpenMP - освободить программиста от подробностей многопоточности, позволяя сосредоточиться на алгоритмических вопросах
  - в OpenMP большинство циклов можно распараллелить с помощью одного простого оператора
  - Распараллеливание циклов хорошо согласуется с SMP: одинаковый код для обработки разных данных
- Актуальные вопросы программирования для многопроцессорных систем:
  - Автоматизация распараллеливания программ
  - Средства отладки
  - Перенос унаследованных приложений на многоядерные процессоры

# Вопросы и задания

1. Напишите псевдокод производителя-потребителя с ограниченным буфером, используя примитивы send(proc_name, &m) и receive(proc_name, &m), где &m – указатель на место в памяти, где может находиться элемент (порция) данных.
2. (У.Столлингс, 2002) Покажите, что сообщения и семафоры обладают эквивалентной функциональностью, для чего реализуйте следующее:

   А. Передачу сообщений с использованием семафоров. *Указание*: сделайте это с помощью буфера для хранения почтовых ящиков, каждый из которых представляет собой массив сообщений.

   Б. Семафоры с использованием передачи сообщений. *Указание*: добавьте в систему синхронизирующий процесс.
3. Почему синхронная схема коммуникации считается более надежной, чем асинхронная?
4. Выведите формулу закона Амдала.
5. Всегда ли логически независимые процессы (т.е., взаимодействие которых ограничивается только конкуренцией за ресурсы) имеет смысл выполнять на разных процессорах? Если не всегда, то когда более экономически эффективно их выполнение на одном и том же процессоре?
6. Если у двух взаимодействующих процессов:

   1) преобладают отношения конкуренции за доступ к критическим ресурсам, то... (а или б?)

   2) преобладают отношения кооперации (сотрудничества), то... (а или б?)

      а) их нужно выполнять на одном и том же процессоре

      б) их нужно выполнять на разных процессорах