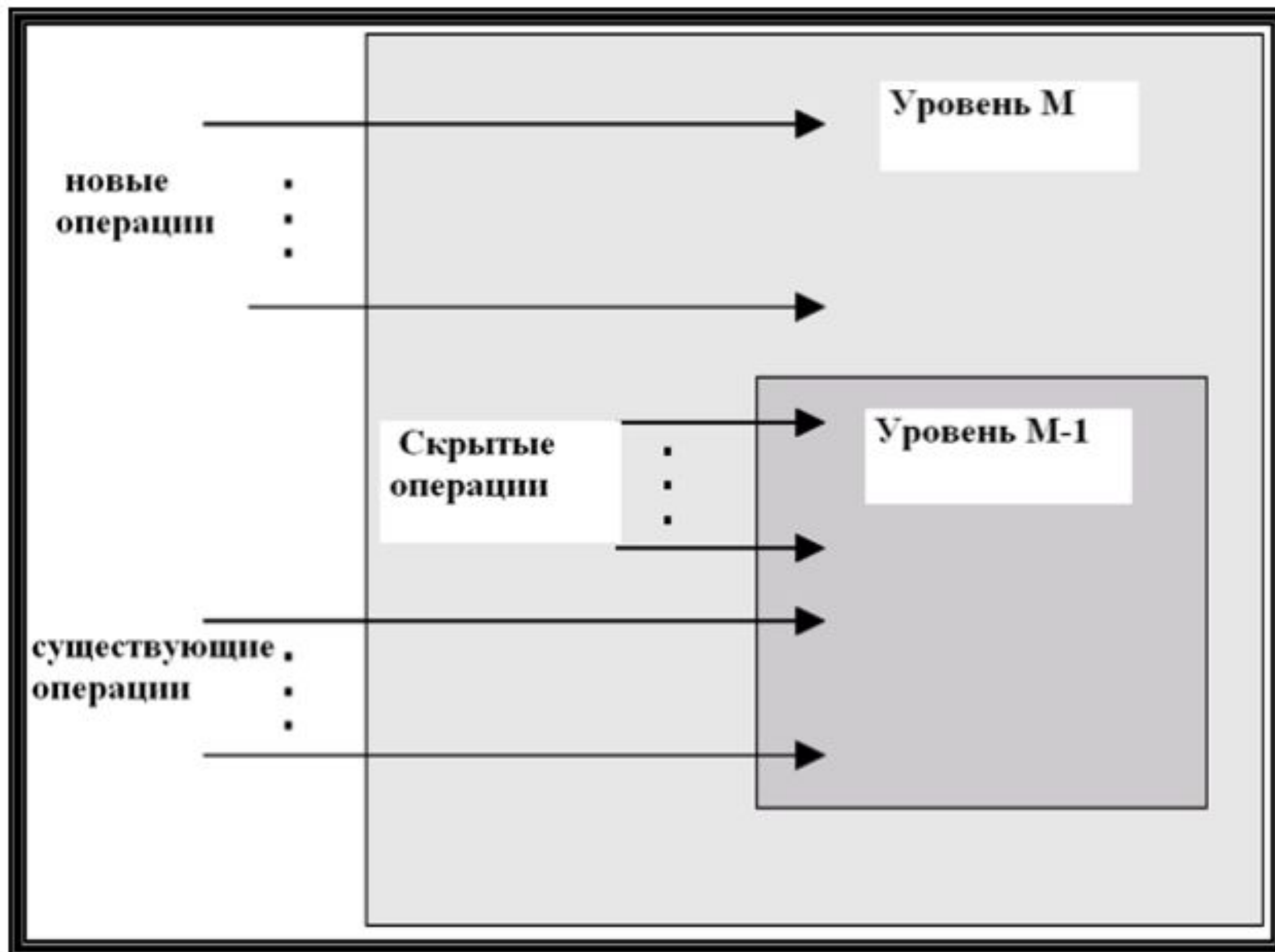
Операционные системы

Уровни абстракции ОС.
ОС с архитектурой микроядра.
Виртуальные машины.
Цели проектирования и разработки ОС.
Генерация ОС .

Уровни абстракции ОС



Структура уровней абстракции OS/2

Структура системы с “микроядром” (micro-kernel)

- **Максимум модулей переносится из ядра ОС в “пользовательское пространство”**
- **Коммуникация выполняется между пользовательскими модулями с помощью передачи сообщений**
- **Преимущества**

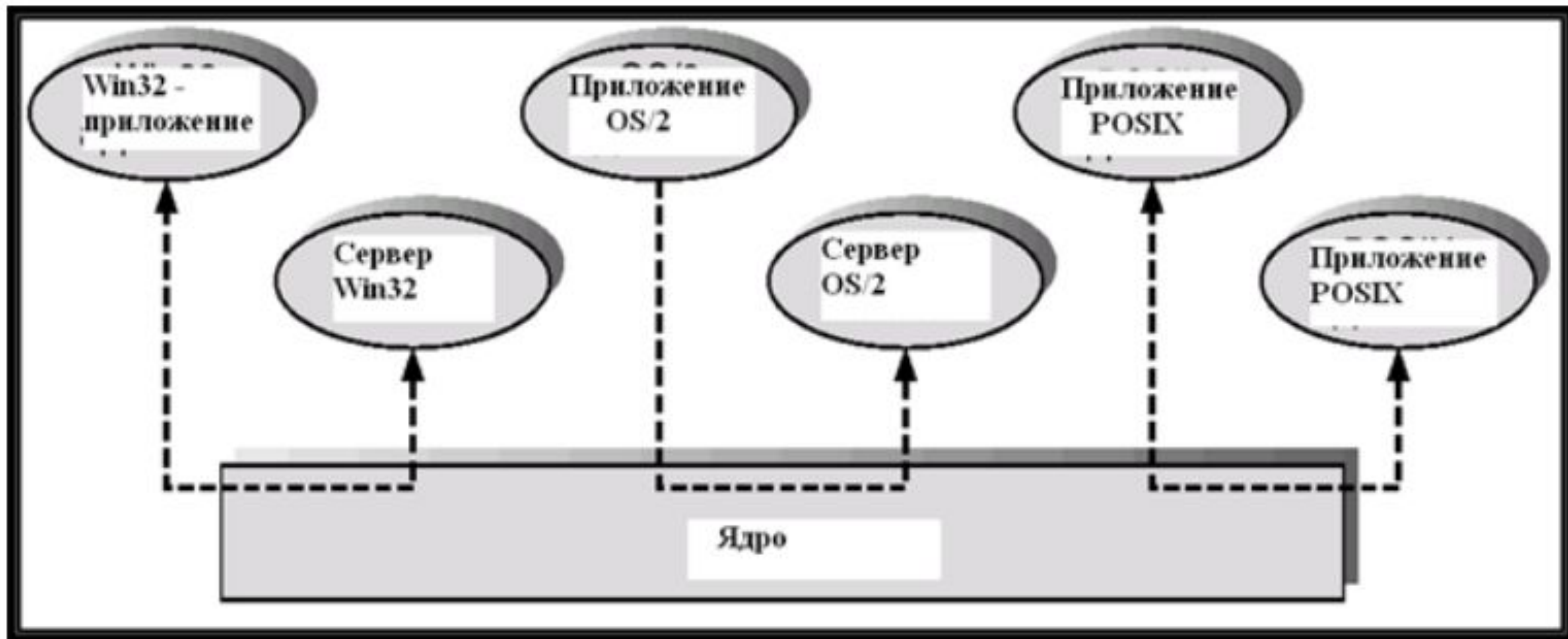
  - микроядро легче расширять

  - легче переносить ОС на новые аппаратные платформы

  - увеличение надежности (большее число программ выполняются в непривилегированном режиме)

  - более безопасно

Клиент-серверная структура Windows NT



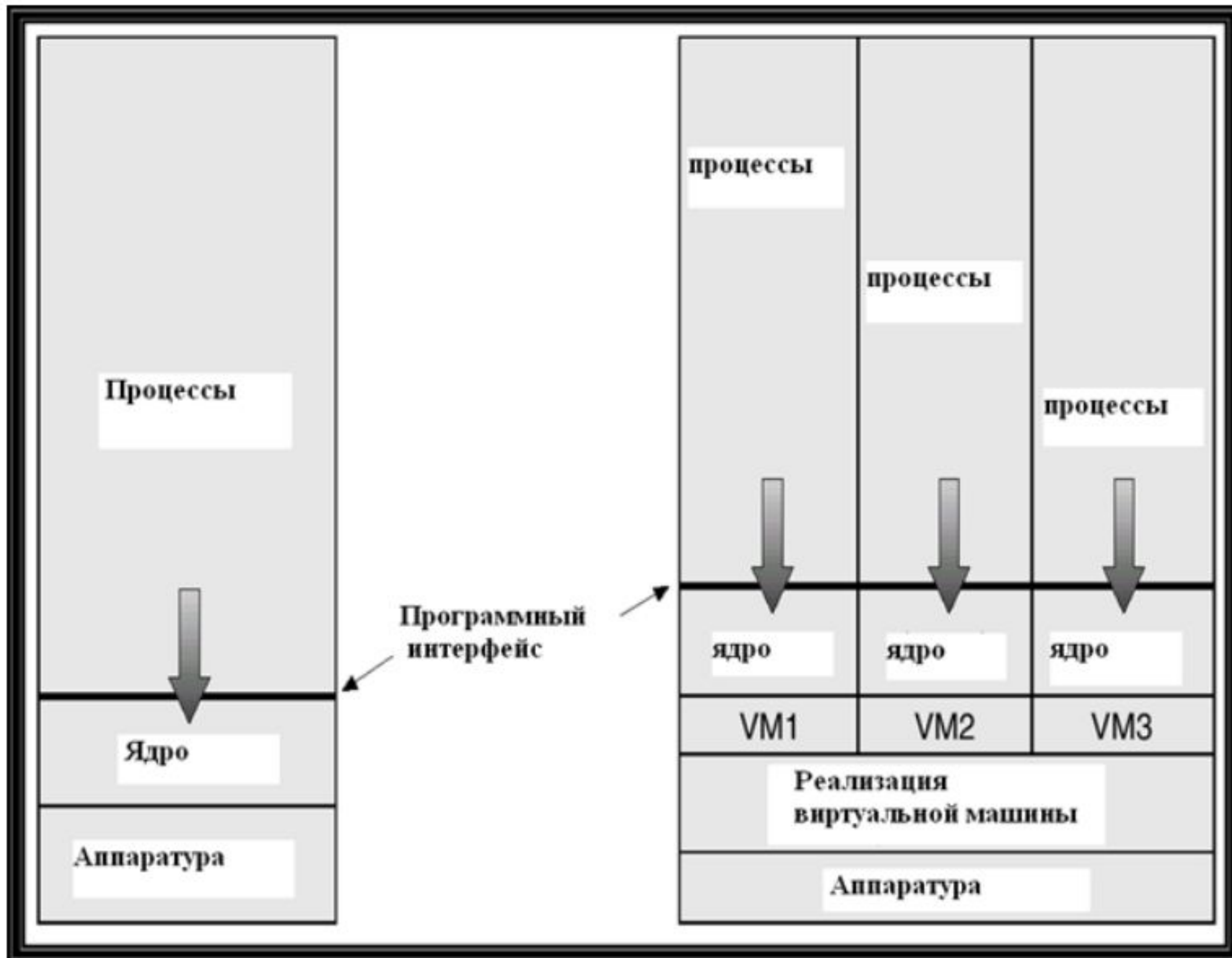
Виртуальные машины

- **Концепция виртуальной машины доводит подход, основанный на уровнях абстракции, до своего логического завершения. Согласно данной концепции, совокупность аппаратуры и ОС трактуется как аппаратура (машина)**
- **Виртуальная машина предоставляет интерфейс, полностью аналогичный интерфейсу обычной машины без базового программного обеспечения**
- **ОС создает иллюзию одновременного исполнения нескольких процессов, каждого в своей (виртуальной) памяти**
- **Пример: система виртуальных машин (SVM) в ОС IBM 360/370, конец 1980-х гг.**

Виртуальные машины (продолжение)

- **Физические ресурсы компьютерной системы разделяются для поддержки нескольких виртуальных машин**
  - **Диспетчеризация процессора создает впечатление, что каждый пользователь имеет свой собственный процессор**
  - **Буферизация (spooling) и файловая система предоставляют виртуальные устройства ввода и вывода**
  - **Терминал обычного пользователя, характерный для режима разделения времени, действует как операторская консоль**

Модели ОС без использования виртуальных машин и на основе виртуальных машин



Преимущества и недостатки виртуальных машин

- **Концепция виртуальной машины обеспечивает полную защиту системных ресурсов , так как каждая виртуальная машина изолирована от других, Однако такая изоляция препятствует совместному использованию ресурсов**
- **Система виртуальных машин – хорошая основа для исследования и разработок в области ОС. Разработка систем выполняется над виртуальной машиной, а не на физической машине, и не нарушает нормального функционирования системы**
- **Концепцию виртуальной машины трудно реализовать, так как трудно адекватно смоделировать используемую машину**

Виртуальная машина Java (JVM)

- **Программы на Java компилируются в платформно-независимый байт-код (bytecode), исполняемый виртуальной машиной Java (JVM).**
- **JVM состоит из:**

  - загрузчика классов (class loader)

  - верификатора классов (class verifier)

  - интерпретатора (runtime interpreter)
- **Just-In-Time (JIT) – компиляторы увеличивают производительность**
- **Аналогичную архитектуру имеет VES (Virtual Execution System) платформы Microsoft.NET**

Виртуальная машина Java

java .class -файлы

загрузчик классов

верификатор

интерпретатор
Java байт-кода

компьютерная система

Цели проектирования и разработки ОС

- ***Цели с точки зрения пользователя*:**

  ОС должна быть удобной в использовании, простой для изучения, надежной, безопасной и быстрой

- ***Цели с точки зрения разработчика ОС*:**

  ОС должна быть несложной для проектирования, реализации и сопровождения, а также гибкой, надежной, свободной от ошибок и эффективной

Механизмы (mechanisms) и политики (policies)

- ***Механизмы*** **– определяют, *каким образом* реализовать функциональность; *политики* - определяют, *что* именно требуется реализовать**
- **Отделение механизма от политики – очень важный принцип; он допускает максимум гибкости, если “политические” решения могут быть изменены впоследствии**

Реализация ОС

- **Традиционно ОС разрабатывались на ассемблере; теперь они могут разрабатываться на языках высокого уровня**
- **Код на языке высокого уровня:**
  - **Может быть разработан быстрее**
  - **Более компактен**
  - **Легче для понимания и отладки**
- **ОС гораздо легче *переносима* на другие аппаратные платформы, если она разработана на языке высокого уровня**

Генерация ОС (SYSGEN)

- **ОС проектируются с целью использования на любой машине из некоторого класса; для каждого конкретного компьютера система должна быть сконфигурирована**
- **Программа SYSGEN получает информацию о специфической конфигурации компьютерной системы**
- ***Загрузка (booting)* – запуск компьютера посредством загрузки ядра ОС**
- ***Программа раскрутки (bootstrap program)* – код, хранящийся в ПЗУ (ROM), который находит ядро ОС, загружает его в память и запускает**

Операционные системы

Управление процессами.

Планирование и диспетчеризация процессов

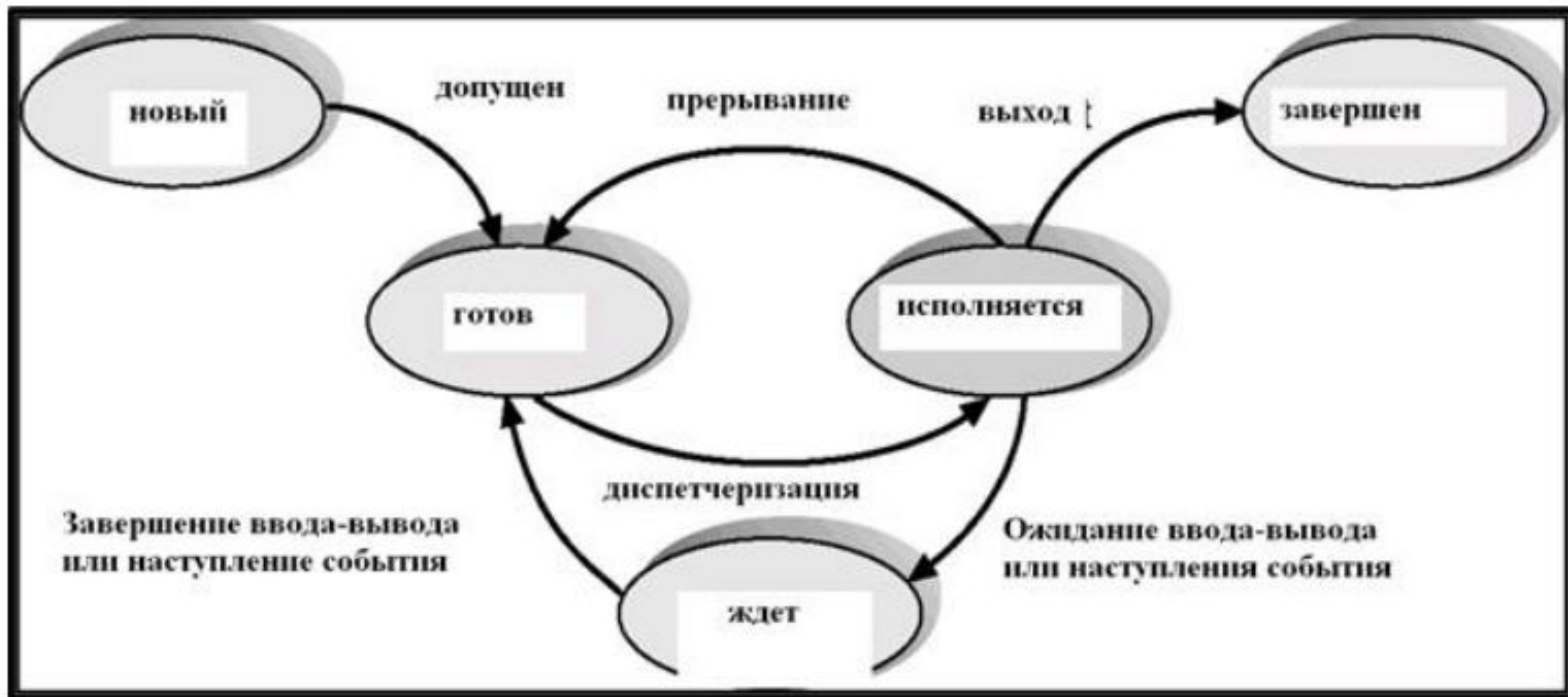
Понятие процесса

- **ОС исполняет множество классов программ:**
  - **Пакетная система (batch system) – задания (jobs)**
  - **Система с разделением времени – пользовательские программы (задачи – tasks)**
- **Во многих учебниках термины “задание” и “процесс” – почти синонимы**
- **Процесс – программа при ее выполнении; он должен выполняться последовательно**
- **Процесс включает:**
  - **Счетчик команд (program counter)**
  - **Стек (stack)**
  - **Секцию данных (data section)**

Состояния процесса

- **При исполнении процесс может изменять свое состояние следующим образом:**
  - ***Новый*** **(new): Процесс создается.**
  - ***Исполняемый*** **(running): Исполняются команды процесса**
  - ***Ожидающий*** **(waiting): Процесс ожидает наступления некоторого события (event)**
  - ***Готовый к выполнению*** **(ready): Процесс ожидает получения ресурсов процессора для его исполнения**
  - ***Завершенный*** **(terminated): Исполнение процесса завершено.**

Диаграмма состояний процесса

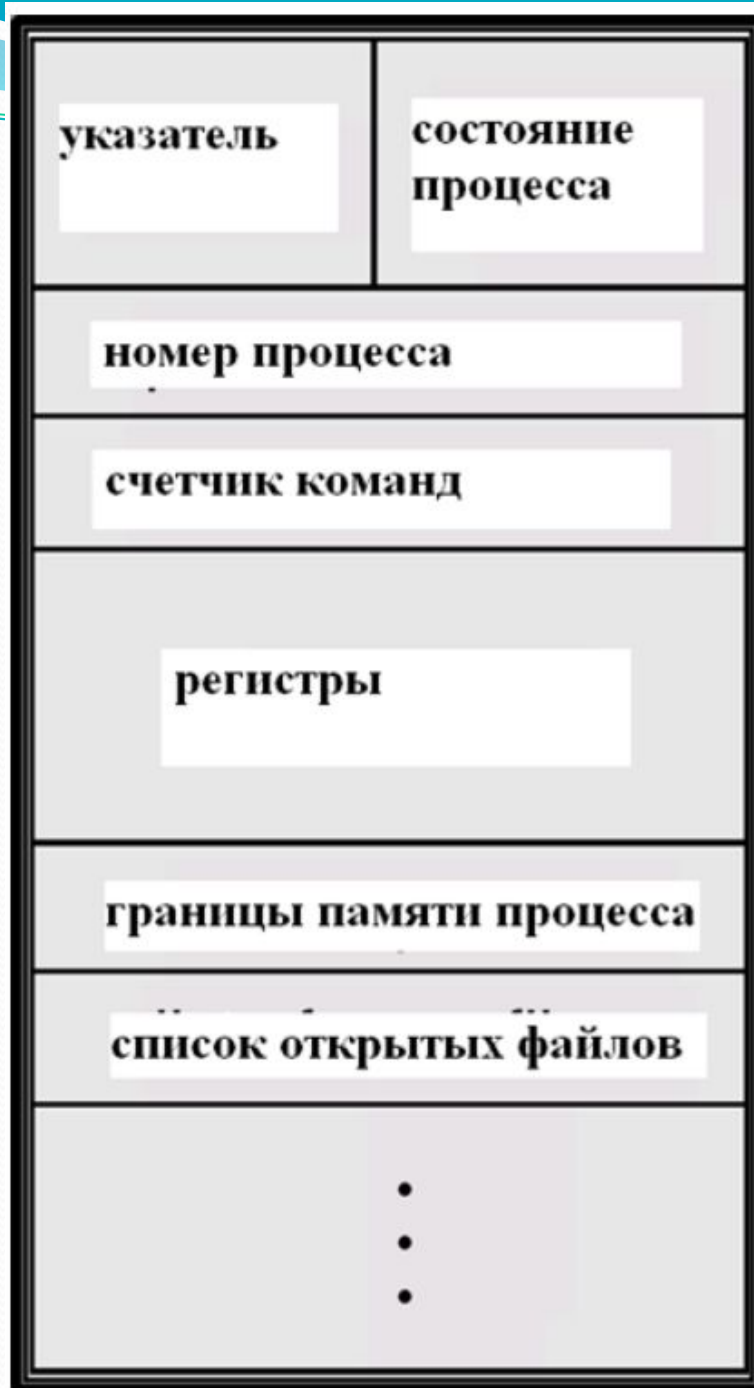


Блок управления процессом (Process Control Block – PCB)

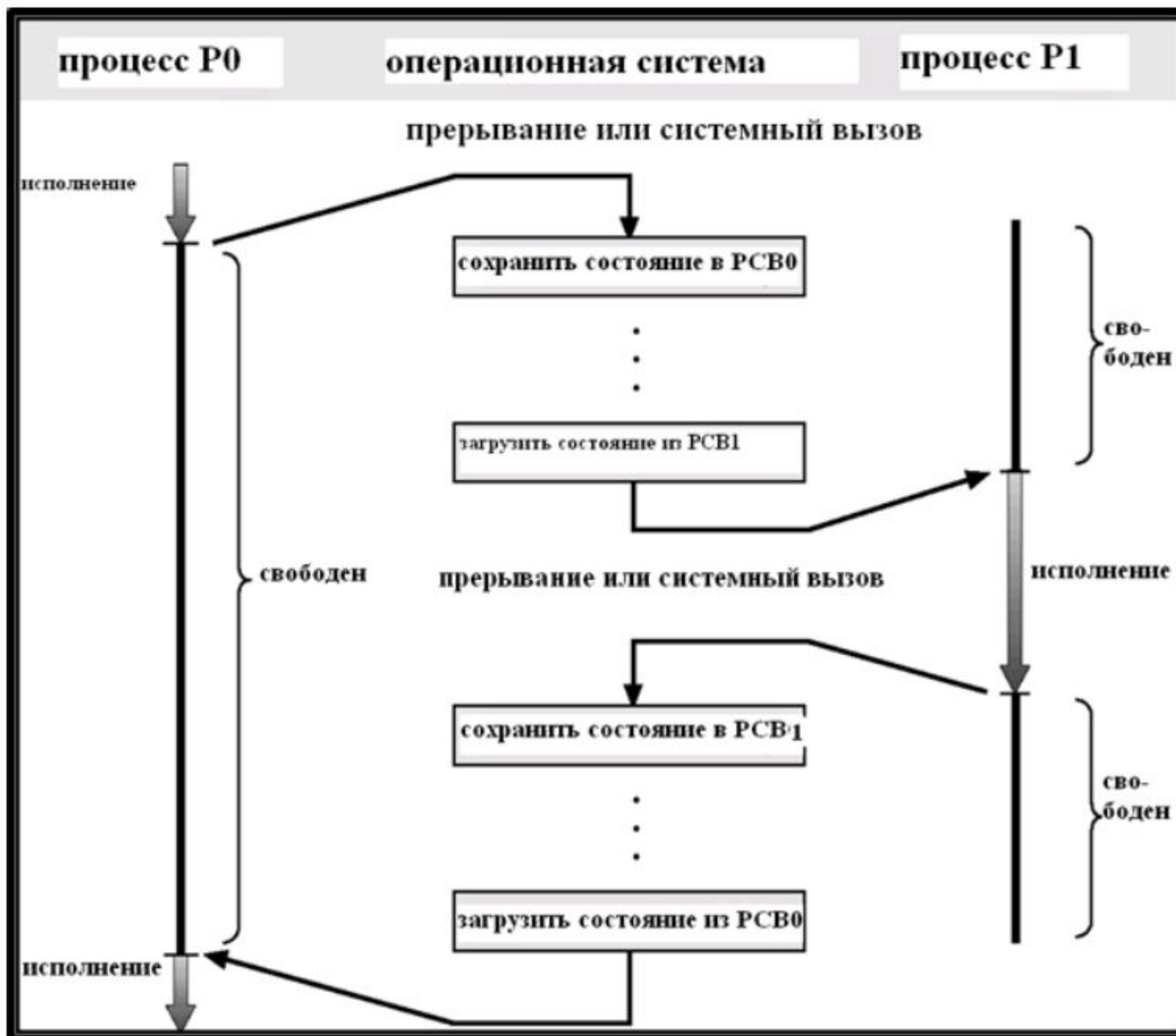
Информация, ассоциируемая с каждым процессом

- **Состояние процесса**
- **Счетчик команд**
- **Регистры процессора**
- **Информация для диспетчеризации процессора**
- **Информация для управления памятью**
- **Статистическая информация**
- **Информация о состоянии ввода-вывода**

Блок управления процессом (PCB)



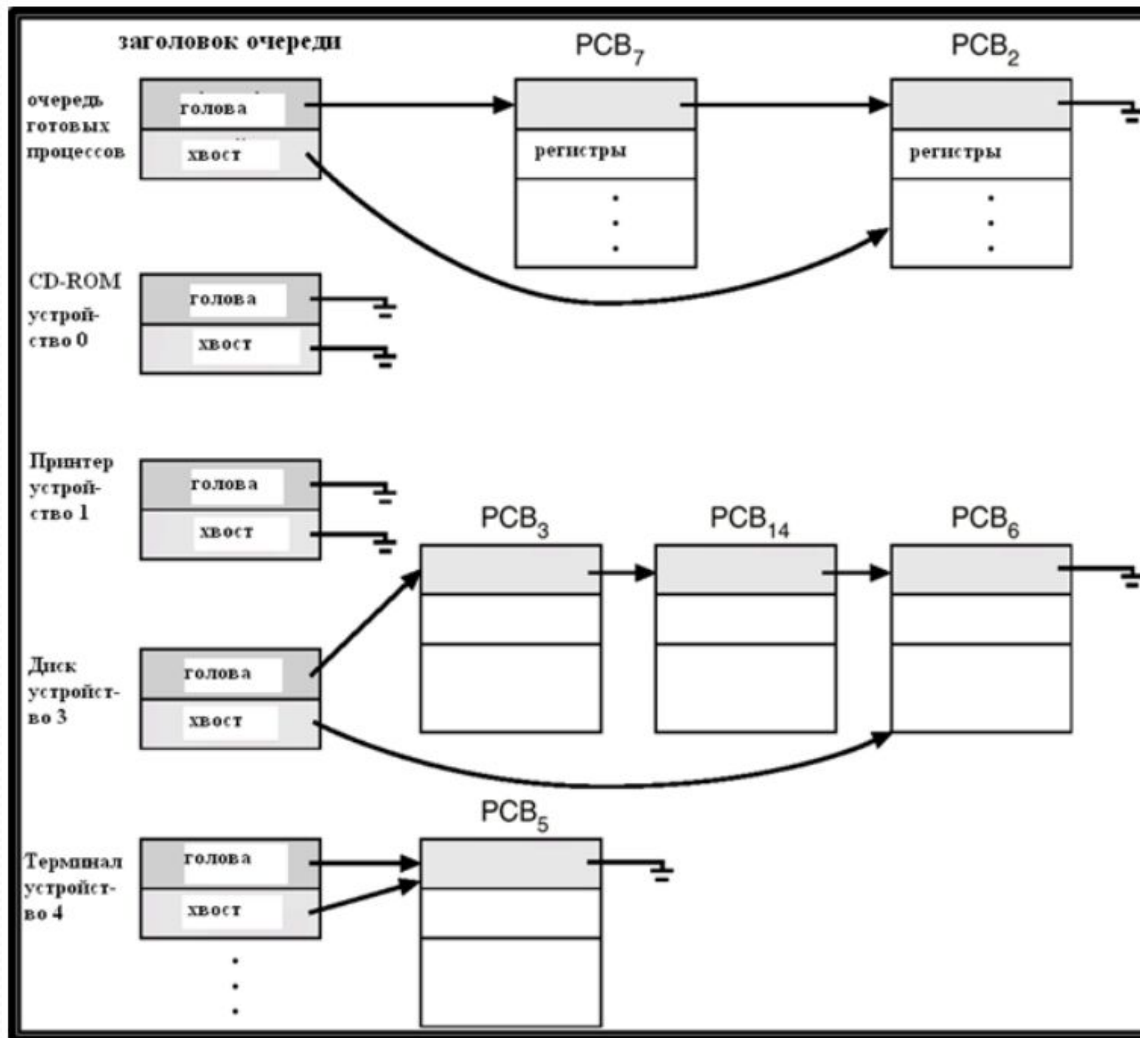
Переключение процессора с одного процесса на другой



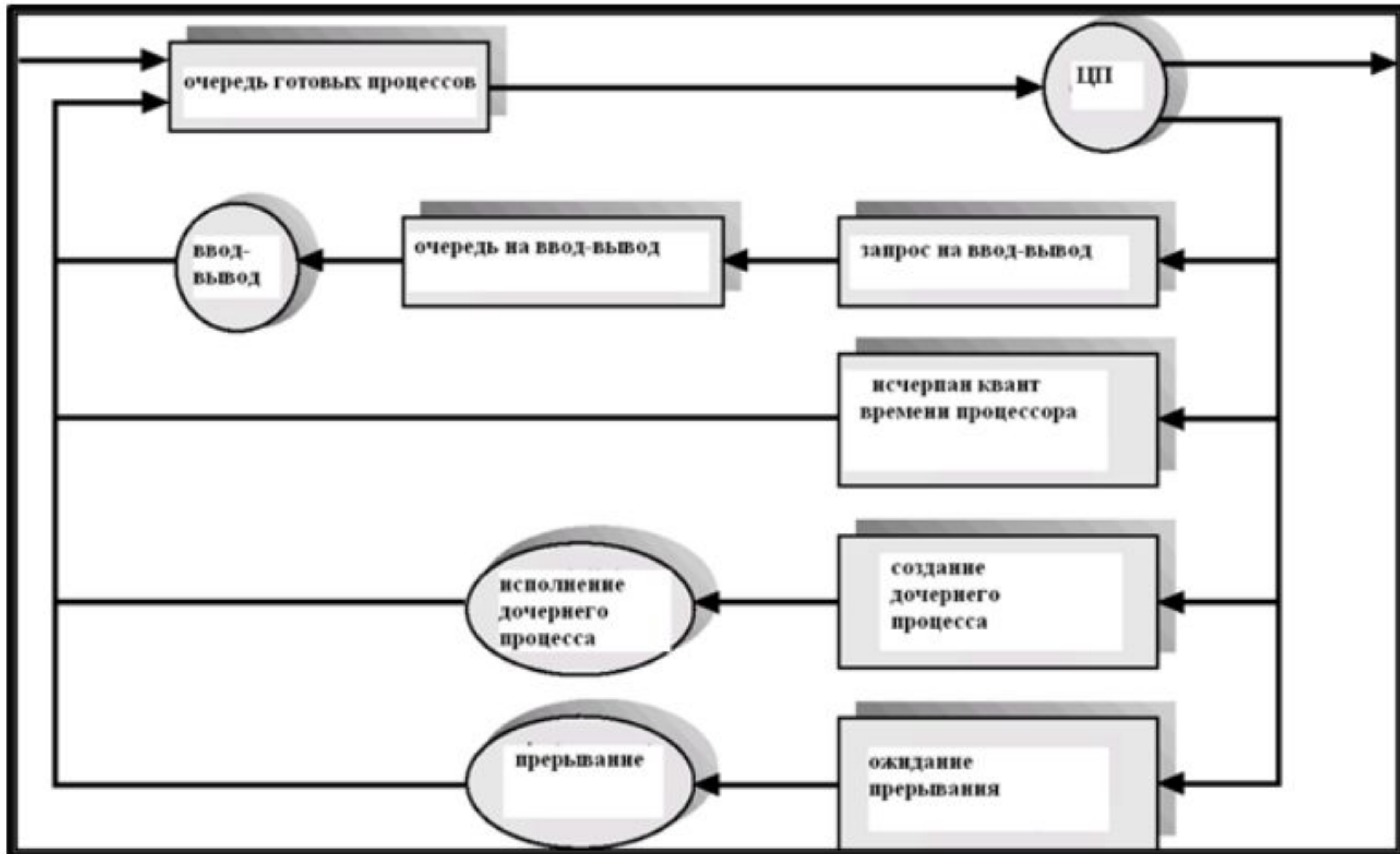
Очереди, связанные с диспетчеризацией процессов

- **Очередь заданий (Job queue) – множество всех процессов в системе**
- **Очередь готовых процессов (Ready queue) – множество всех процессов, находящихся в основной памяти и готовых к выполнению**
- **Очередь ожидающих ввода-вывода (Device queues) – множество процессов, ожидающих результата работы устройства ввода-вывода**
- **Процессы мигрируют между различными очередями**

Очередь готовых процессов и очереди для различных устройств ввода-вывода



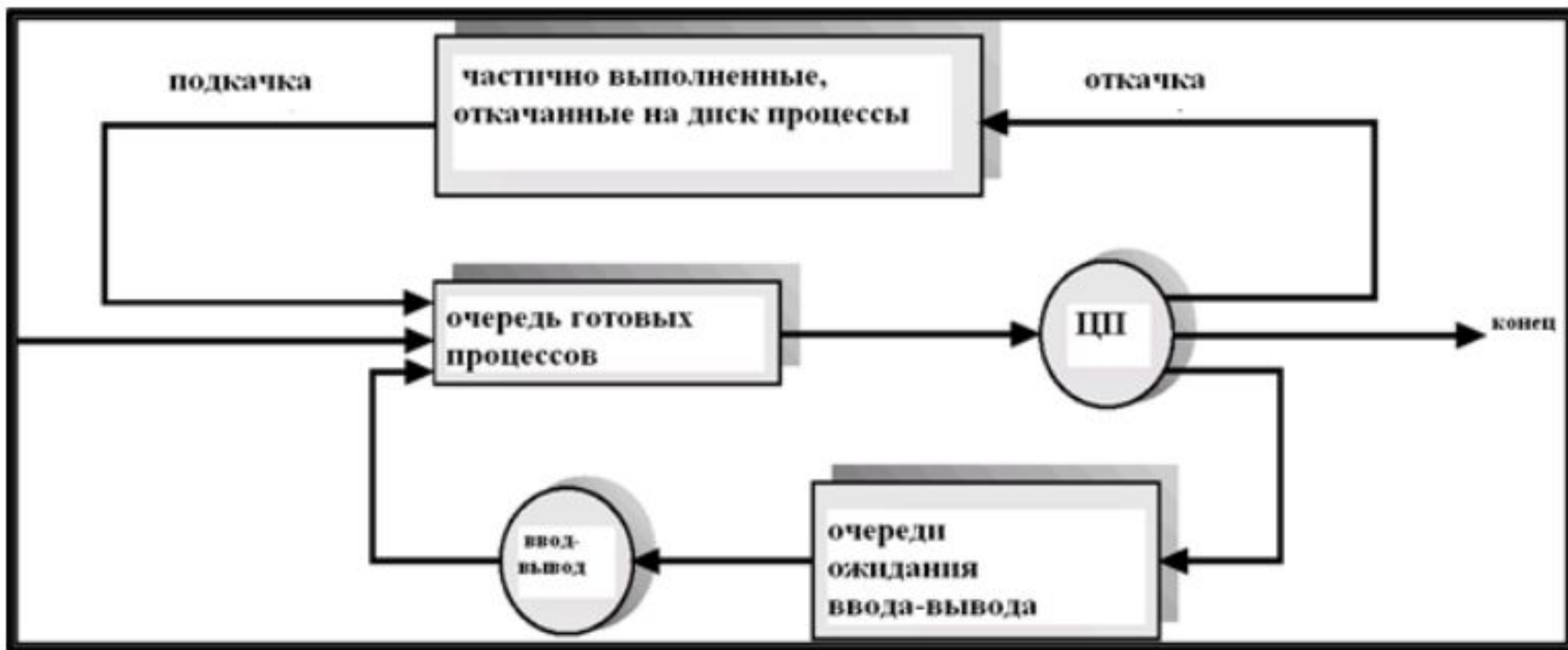
Графическое представление диспетчеризации процессов



Диспетчеры

- **Долговременный диспетчер (диспетчер заданий) – определяет, какие процессы должны быть перемещены в очередь готовых процессов**
- **Кратковременный диспетчер (диспетчер процессора) – определяет, какие процессы должны быть выполнены следующими и каким процессам должны быть предоставлены процессоры.**

Добавление промежуточного диспетчера



Диспетчеры (продолжение)

- **Кратковременный диспетчер вызывается очень часто (в течение ближайших миллисекунд) => должен быть очень быстрым**
- **Долговременный диспетчер вызывается относительно редко (минуты, секунды) => может быть сравнительно медленным**
- **Именно долговременный диспетчер определяет *степень (коэффициент) мультипрограммирования***
- **Процессы можно описать как:**
  - **Ориентированные на ввод-вывод (I/O-*bound*) – тратят больше времени на ввод-вывод, чем на вычисления; расходуют много коротких квантов процессорного времени**
  - ***Ориентированные на использование процессора (CPU-bound*) – тратят основное время на вычисления; расходуют небольшое число долговременных квантов процессорного времени**

Переключение контекста процесса (context switch)

- **Когда процессор переключается на другой процесс, система должна сохранить состояние старого процесса и загрузить сохраненное состояние для нового процесса**
- **Переключение контекста относится к накладным расходам (overhead); система не выполняет никаких полезных действий при переключении с одного процесса на другой**
- **Время зависит от аппаратной поддержки.**
- **Пример: “Эльбрус” – контекстное переключение – одна команда *СМСТЕК* (сменить стек, т.е. переключиться с одного облегченного процесса на другой)**

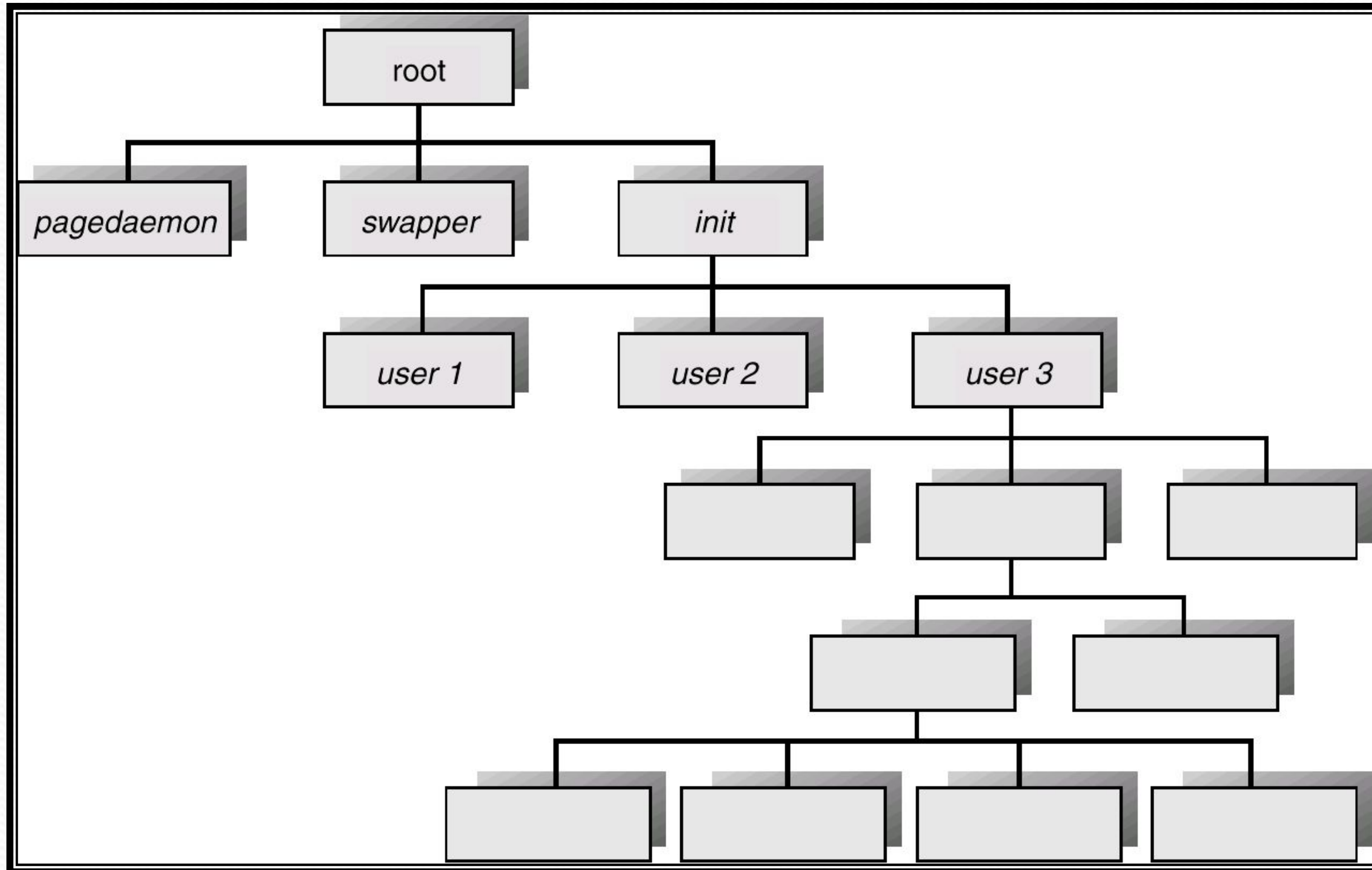
Создание процесса

- **Процесс-родитель создает дочерние процессы, которые, в свою очередь, создают другие процессы, тем самым формируя *дерево процессов***
- **Разделение ресурсов**
  - **Процесс-родитель и дочерние процессы разделяют все ресурсы**
  - **Дочерние процессы разделяют подмножество ресурсов процесса-родителя**
  - **Процесс-родитель и дочерний процесс не имеют общих ресурсов**
- **Исполнение**
  - **Процесс-родитель и дочерние процессы исполняются совместно**
  - **Процесс-родитель ожидает завершения дочерних процессов**

Создание процесса (продолжение)

- Адресное пространство
  - Дочернего процесса копирует адресное пространство процесса-родителя
  - У дочернего процесса имеется программа, загруженная в него
- UNIX:
  - fork – системный вызов, создающий новый процесс
  - exec (execve) – системный вызов, используемый после fork, с целью замены пространства памяти процесса новой программой

Дерево процессов в системе UNIX



Уничтожение процесса

- **Процесс исполняет заключительный оператор и обращается к ОС для своей ликвидации (exit).**
  - **Передача данных от дочернего процесса процессу-родителю (wait).**
  - **Ресурсы процесса освобождаются операционной системой**
- **Процесс-родитель может уничтожить дочерние процессы (abort).**
  - **Дочерний процесс превысил выделенные ему ресурсы**
  - **Решения задачи, порученной дочернему процессу, больше не требуется**
  - **Происходит выход из процесса-родителя**
    - **ОС не допускает продолжения исполнения дочернего процесса, если его процесс-родитель уничтожается**
    - **“Каскадное” уничтожение процессов**

Операционные системы

Методы взаимодействия процессов

Взаимодействующие (cooperating) процессы

- ***Независимый*** **процесс – не может влиять на исполнение других процессов и испытывать их влияние.**
- ***Взаимодействующий (совместный)*** **процесс – может влиять на исполнение других процессов или испытывать их влияние**
- **Преимущества взаимодействующих процессов**
  - **Совместное использование данных**
  - **Ускорение вычислений**
  - **Модульность**
  - **Удобство**

Виды процессов

- *Подчиненный* – зависит от процесса-родителя; уничтожается при его уничтожении; процесс-родитель должен ожидать завершения всех подчиненных процессов
- *Независимый* – не зависит от процесса-родителя; исполняется независимо от него (например, *процесс-демон: cron, smbd* и др.)
- *Сопроцесс (co-process, co-routine)* – хранит свое текущее *локальное управление* (program counter); взаимодействует с другим сопроцессом *Q* с помощью операций *resume (Q)*. Операция *detach* переводит сопроцесс в пассивное состояние (SIMULA-67). *Пример*: взаимодействие *итератора* с циклом

Проблема “производитель-потребитель” (producer – consumer)

- **Одна из парадигм взаимодействия процессов: процесс-производитель (*producer*) генерирует информацию, которая используется процессом-потребителем (*consumer*)**
  - ***Неограниченный буфер (unbounded-buffer)* – на размер используемого буфера практически нет ограничений**
  - ***Ограниченный буфер (bounded-buffer)* – предполагается определенное ограничение размера буфера**
  - **Схема с ограниченным буфером, с точки зрения security, представляет опасность *атаки “buffer overruns”*. При заполнении буфера необходимо проверять его *размер*.**

Ограниченный буфер – реализация с помощью общей памяти

- **Общие данные**

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- **Решение правильно, но могут использоваться только (BUFFER_SIZE-1) элементов**

Ограниченный буфер: процесс-производитель

```
item nextProduced;

while (1) {
  while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
}
```

Ограниченный буфер: процесс-потребитель

```
item nextConsumed;

while (1) {
  while (in == out)
        ; /* do nothing */
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
}
```

Взаимодействие процессов

- **Механизм для коммуникации процессов и синхронизации их действий.**
- ***Система сообщений*** **– процессы взаимодействуют между собой без обращений к общим переменным.**
- **Средство взаимодействия между процессами (IPC facility) обеспечивает две операции:**
  - ***send*** **(*message*) – размер сообщения постоянный или переменный**
  - ***receive*** **(*message*)**
- **Если *P* и *Q* требуется взаимодействовать между собой, им необходимо:**
  - **Установить связь (communication link) друг с другом**
  - **Обменяться сообщениями вида send/receive**
- **Реализация связи**
  - **Физическая (общая память, аппаратная шина)**
  - **Логическая (например, логические свойства)**

Реализация взаимодействия процессов

- **Как устанавливается связь?**
- **Можно ли установить связь более чем двух процессов?**
- **Сколько связей может быть установлено между двумя заданными процессами?**
- **Какова пропускная способность линии связи?**
- **Является ли длина сообщения по линии связи постоянной или переменной?**
- **Является ли связь ненаправленной или двунаправленной (дуплексной)?**

Прямая связь (direct communication)

- **Процессы именуют друг друга явно:**
  - **send** (***P, message***) – послать сообщение процессу P
  - **receive**(***Q, message***) – получить сообщение от процесса Q
- **Свойства линии связи**
  - **Связь устанавливается автоматически.**
  - **Связь ассоциируется только с одной парой взаимодействующих процессов.**
  - **Между каждой парой процессов всегда только одна связь.**
  - **Связь может быть ненаправленной, но, как правило, она двунаправленная.**

Косвенная связь (indirect communication)

- **Сообщения направляются и получаются через почтовые ящики (порты) – mailboxes; ports**
  - **Каждый почтовый ящик имеет уникальный идентификатор.**
  - **Процессы могут взаимодействовать, только если они имеют общий почтовый ящик.**
- **Свойства линии связи**
  - **Связь устанавливается, только если процессы имеют общий почтовый ящик**
  - **Связь может быть установлена со многими процессами.**
  - **Каждая пара процессов может иметь несколько линий связи.**
  - **Связь может быть ненаправленной или двунаправленной.**

Косвенная связь

- **Операции**
  - **Создать новый почтовый ящик**
  - **Отправить (принять) сообщение через почтовый ящик**
  - **Удалить почтовый ящик**
- **Основные операции**:

send **(*A*, *message*) – послать сообщение в почтовый ящик A**

receive **(*A*, *message*) – получить сообщение из почтового ящика A**

Косвенная связь

- **Использование общего почтового ящика**
  - P_1, P_2, и P_3 используют почтовый ящик A.
  - P_1, посылает сообщение; P_2 и P_3 принимают.
  - Кто получает сообщение?
- **Решения**
  - Ограничить связь только двумя процессами.
  - Разрешить только одному процессу в каждый момент исполнять операцию получения
  - Разрешить системе произвольным образом определить получателя
  - Отправитель нотифицируется, кто является получателем.

Синхронизация при косвенной связи

- **Передача сообщений может выполняться с блокировкой или без блокировки**
- **Передача с блокировкой - синхронная**
- **Передача без блокировки - асинхронная**
- **Основные операции send и receive могут быть с блокировкой или без блокировки**

Буферизация

- **С коммуникационной линией связывается *очередь сообщений*, реализованная одним из трех способов:**

1. **Нулевая емкость – 0 сообщений**
   Отправитель должен ждать получателя (рандеву - rendezvous).
2. **Ограниченная емкость – конечная длина очереди: *n* сообщений (предотвратить опасность атаки "buffer overruns"!)**

   Отправитель должен ждать, если очередь заполнена.
3. **Неограниченная емкость – бесконечная длина.**
   Получатель никогда не ждет.

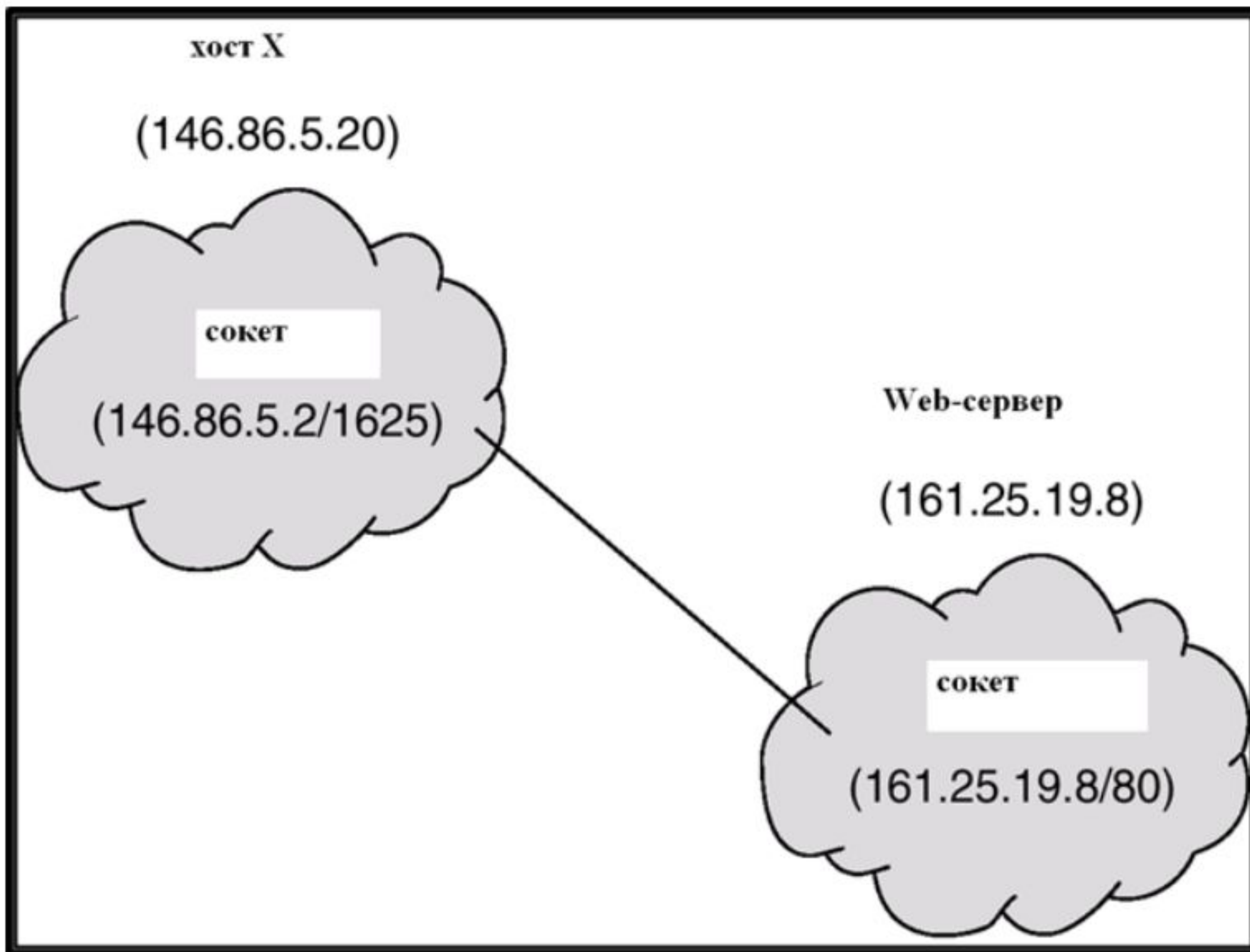
Клиент-серверная взаимосвязь

- **Сокеты (Sockets)**
- **Удаленные вызовы процедур (Remote Procedure Calls – RPC)**
- **Удаленные вызовы методов (Remote Method Invocation – RMI) : Java**

Сокеты (Sockets)

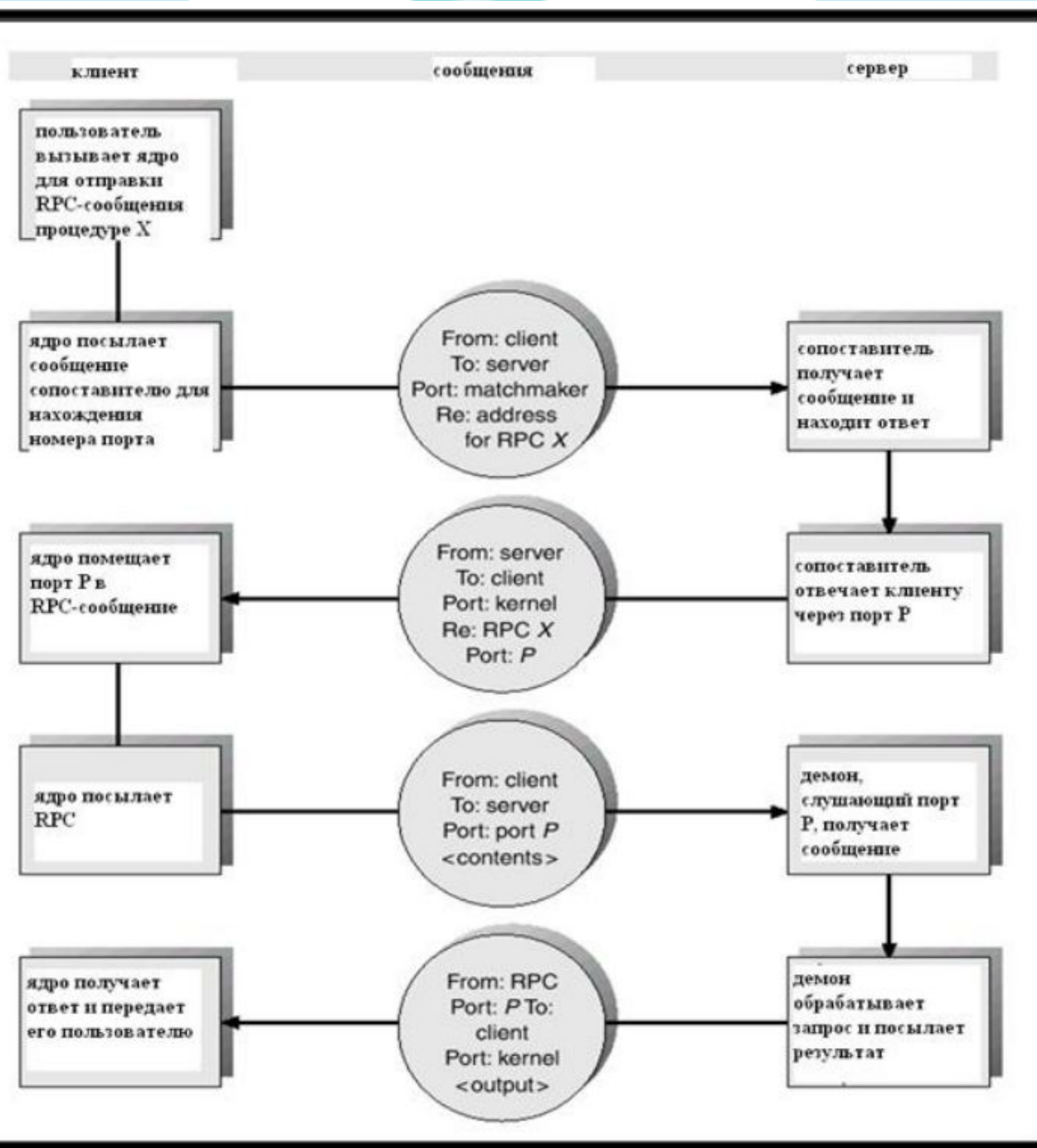
- **Впервые были реализованы в UNIX BSD 4.2**
- **Сокет можно определить как отправную (конечную) точку для коммуникации - *endpoint for communication*.**
- **Конкатенация IP-адреса и порта**
- **Сокет 161.25.19.8:1625 ссылается на порт 1625 на машине (хосте) 161.25.19.8**
- **Коммуникация осуществляется между парой сокетов**

Взаимодействие с помощью сокетов



Удаленные вызовы процедур (RPC)

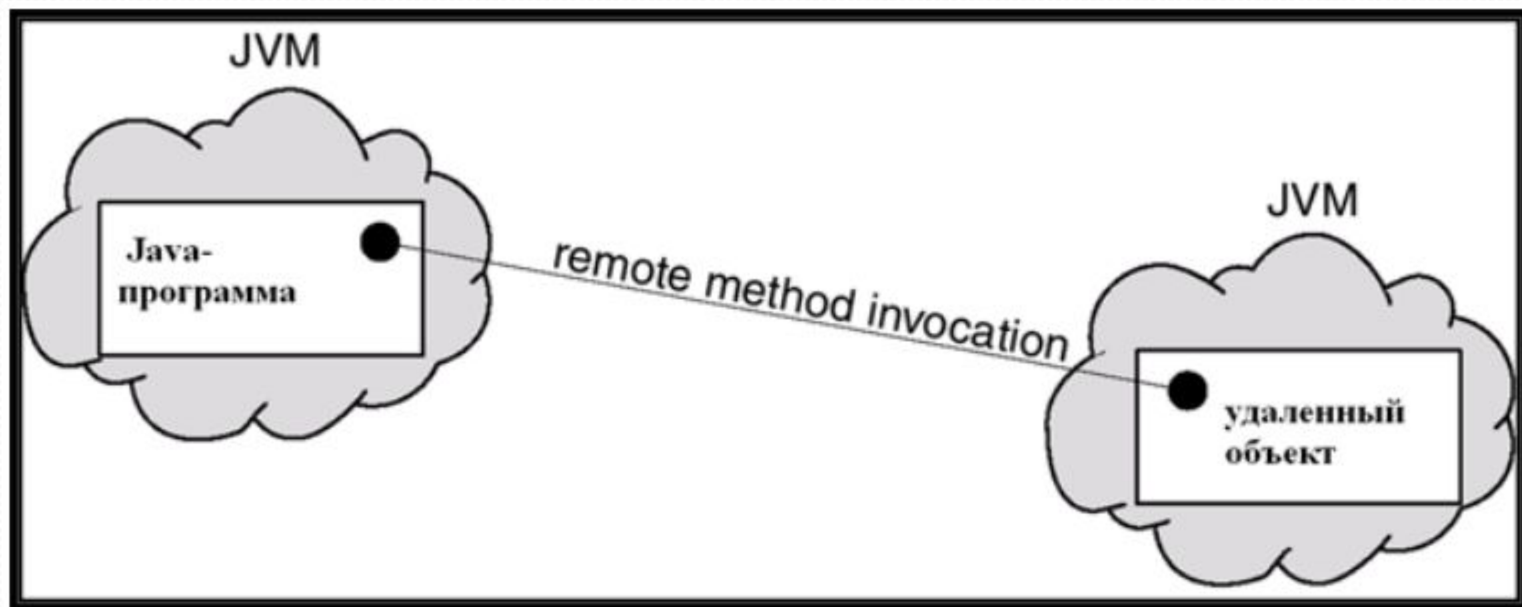
- **RPC впервые предложен фирмой Sun и реализован в ОС Solaris**
- **Удаленный вызов процедуры (RPC) – абстракция вызова процедуры между процессами в сетевых системах**
- **Заглушки (Stubs) – proxy в клиентской части для фактической процедуры, находящейся на сервере**
- **Заглушка в клиентской части находит сервер и выстраивает (*marshals*) параметры.**
- **Заглушка в серверной части принимает это сообщение, распаковывает параметры, преобразует их к нормальному виду и выполняет процедуру на сервере**



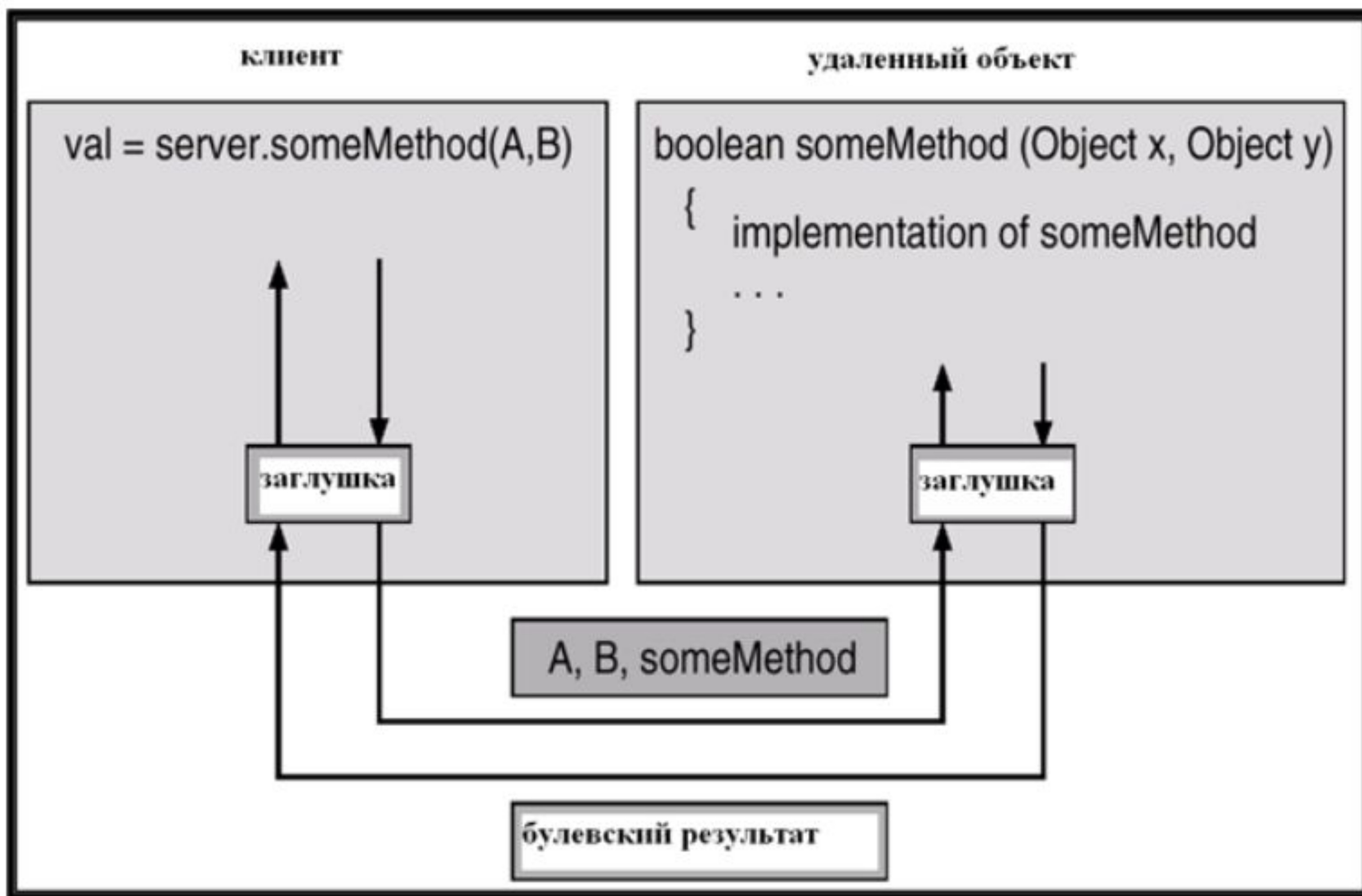
Исполнение RPC

Удаленный вызов метода (RMI) - Java

- **Remote Method Invocation (RMI) – механизм в Java-технологии, аналогичный RPC**
- **RMI позволяет Java-приложению на одной машине вызвать метод удаленного объекта.**



Выстраивание параметров (marshaling)

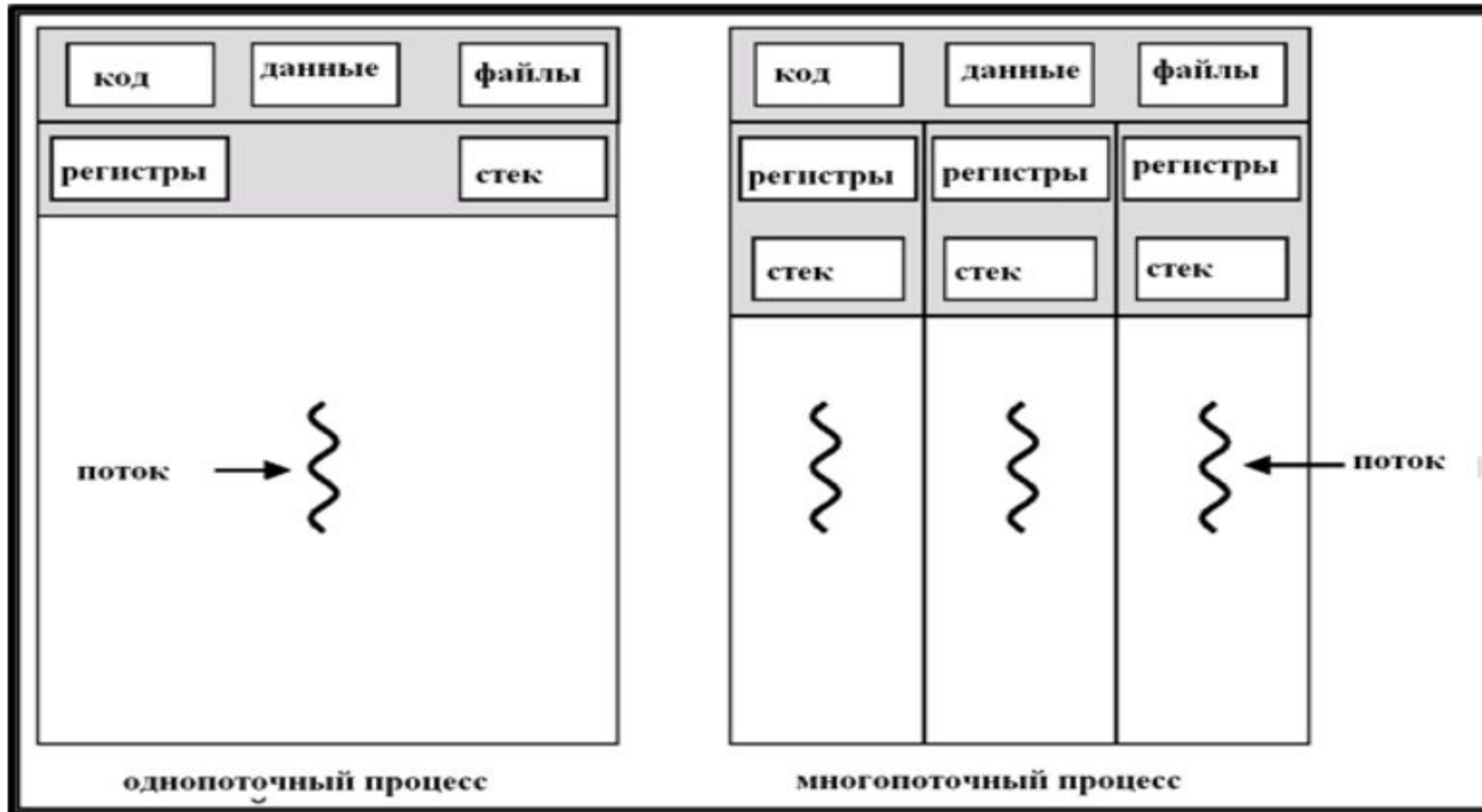


Операционные системы

Потоки.

Однопоточные и многопоточные процессы

Последовательный (однопоточный) процесс – это процесс, который имеет только один поток управления (control flow), характеризующийся изменением его счетчика команд.
Поток (thread) – это запускаемый из некоторого процесса особого рода параллельный процесс, выполняемый в том же адресном пространстве, что и процесс-родитель.



Преимущества многопоточности

- **Увеличение скорости (по сравнению с использованием обычных процессов). Многопоточность основана на использовании *облегченных процессов (lightweight processes)*, работающих в общем пространстве виртуальной памяти**
- **Использование общих ресурсов**
- **Экономия**
- **Использование мультипроцессорных архитектур**

История многопоточности

- “Эльбрус-1” (1979) : концепция процесса соответствовала ***облегченному процессу*** в современном понимании (***процесс*** <-> ***стек***)
- UNIX: конец 80-х – начало 90-х гг. : AT&T, Solaris)
- Windows NT – середина 90-х гг.
- В разных операционных системах API для многопоточности существенно отличаются
- Б. Страуструп ***не*** включил многопоточность в С++
- Java (1995) : впервые многопоточность реализована на уровне языка + core API
- .NET (2000) : многопоточность – фактически развитие идей Java

Пользовательские потоки (user threads)

- **Управление потоками реализовано через библиотеку потоков пользовательского уровня**
- **Примеры**
  - POSIX *Pthreads*
  - Mac *C-threads*
  - Solaris *threads*

Потоки ядра (kernel threads)

- **Поддержаны и используются на уровне ядра ОС**

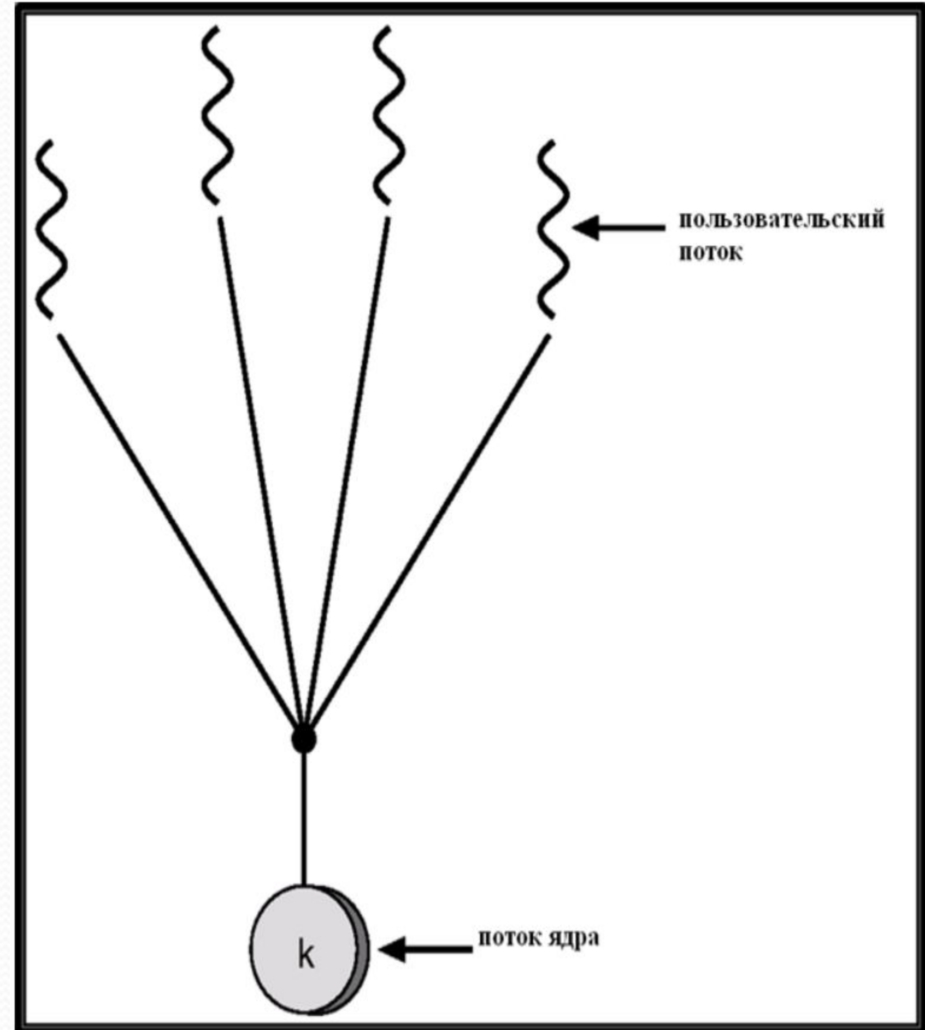
- **Примеры**
  - Windows 95/98/NT/2000
  - Solaris
  - Tru64 UNIX
  - BeOS
  - Linux

Модели многопоточности (каким образом пользовательские потоки отображаются в потоки ядра?)

- **Много / Один (Many-to-One)**
- **Один / Один (One-to-One)**
- **Много / Много (Many-to-Many)**

Модель “много / один”

- **Несколько потоков пользовательского уровня отображаются в один системный поток**
- **Используется в системах, которые не поддерживают множественные системные потоки**

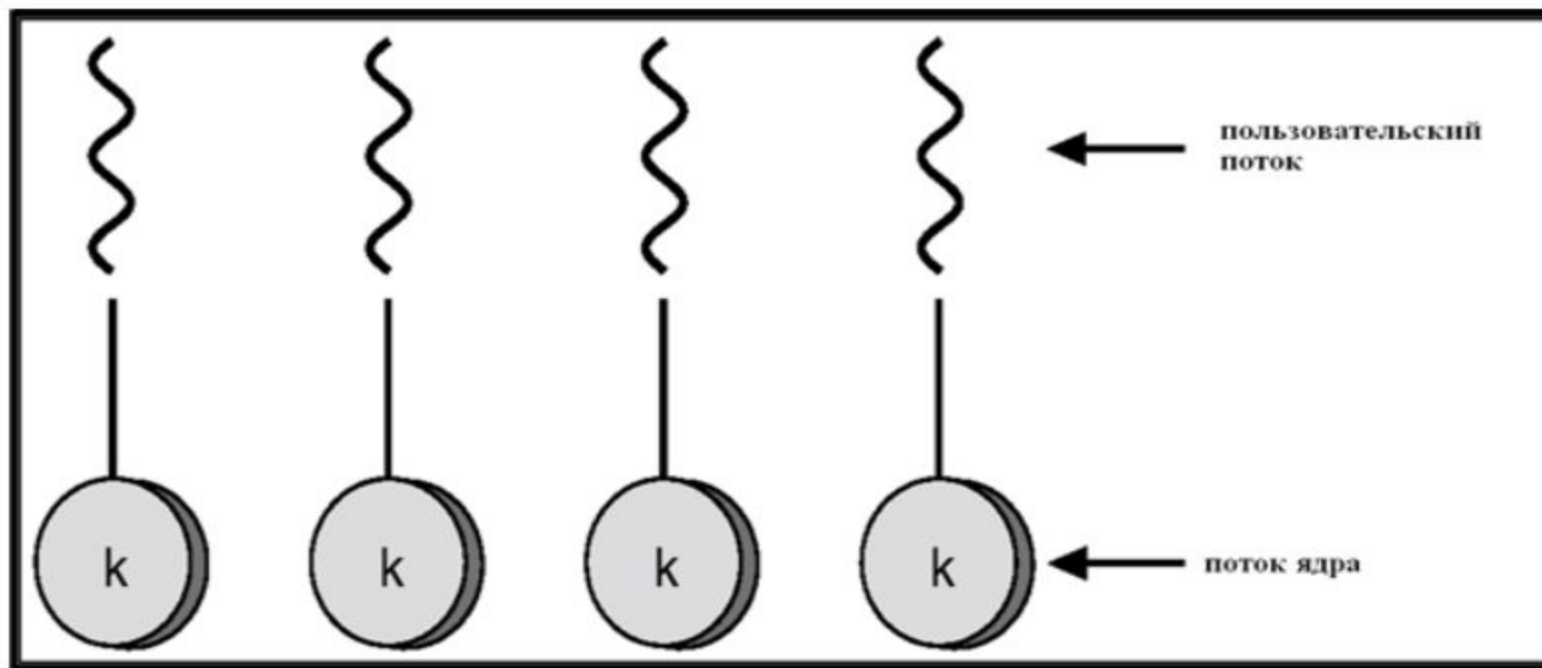


Модель "один / один"

- **Каждый поток пользовательского уровня отображается в один системный поток**
- **Примеры**

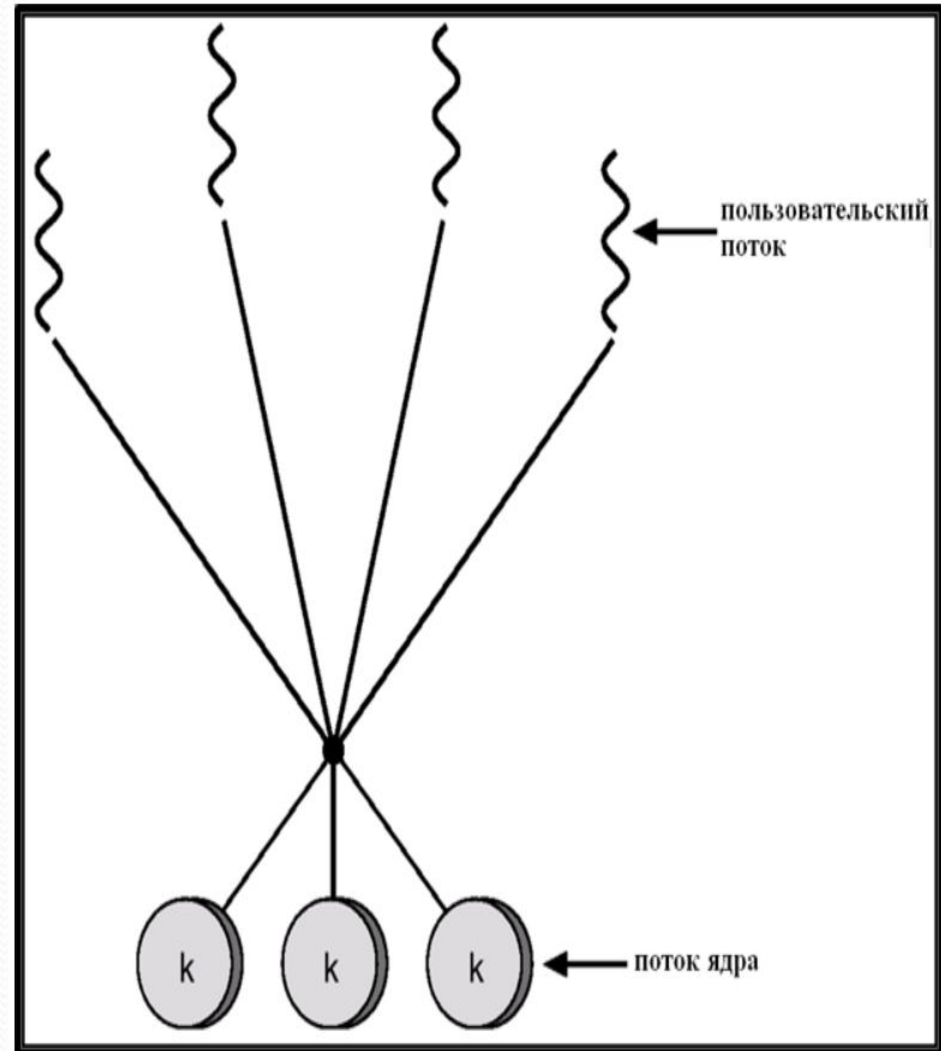
- Windows 95/98/NT/2000/XP/2003/2008/7

- OS/2



Модель “много / много”

- **Допускает, чтобы несколько потоков пользовательского уровня могли отображаться в несколько системных потоков**
- **Позволяет ОС создавать достаточно большое число системных потоков.**
- **Solaris 2.x**
- **Windows NT/2000 с пакетом *ThreadFiber***



Многопоточность – весьма сложная, еще не полностью изученная и, тем более, не полностью формализованная область.

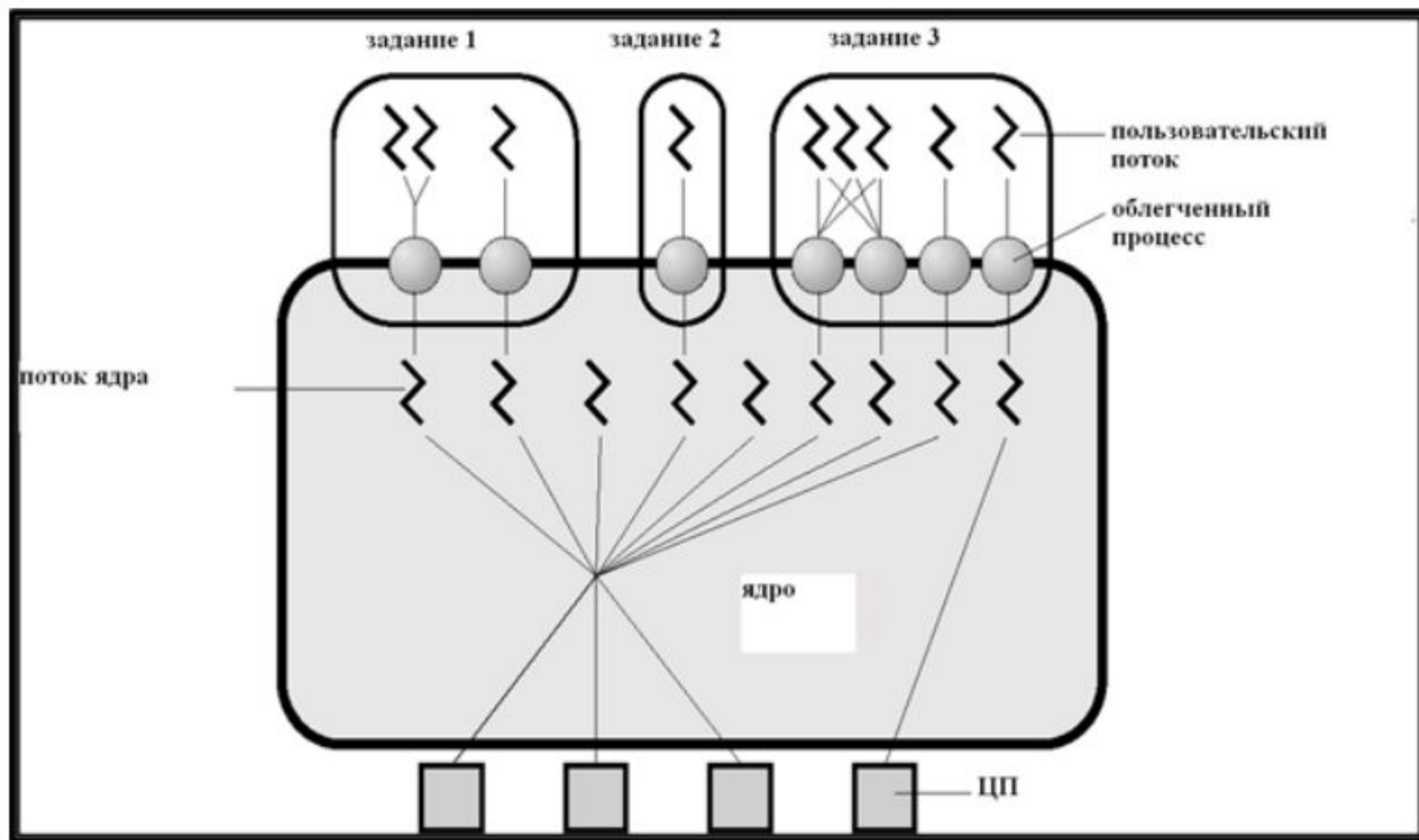
Проблемы многопоточности

- **Семантика системных вызовов fork() и exec()**
- **Прекращение потоков**
- **Обработка сигналов**
- **Группы потоков**
- **Локальные данные потока (thread-local storage)**
- **Синхронизация потоков**
- **Тупики (deadlocks) и их предотвращение**

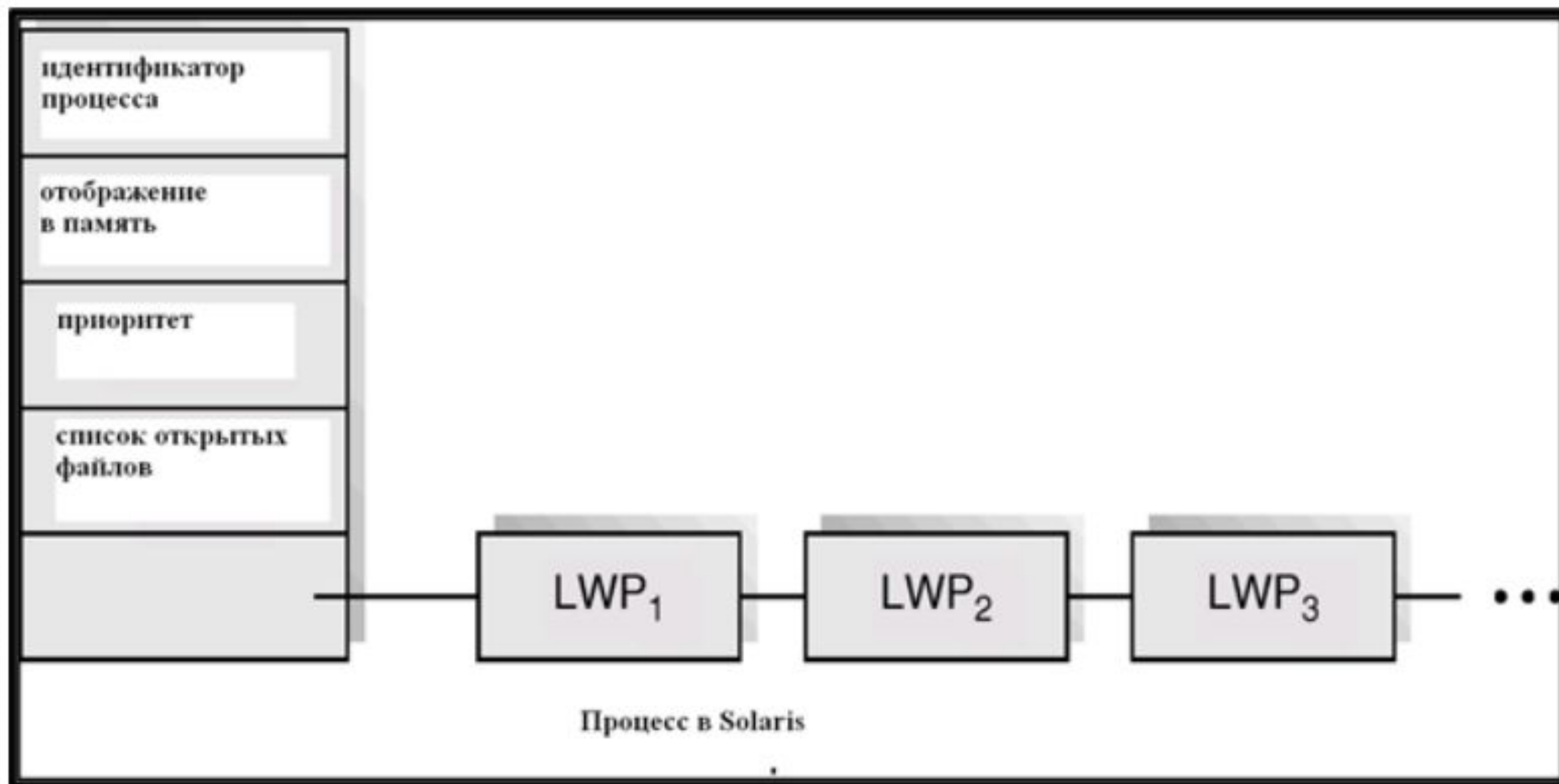
Потоки POSIX (Pthreads)

- **POSIX – Portable Operating Systems Interface of uniX kind**
- **Стандарт POSIX (IEEE 1003.1c) API для создания и синхронизации потоков**
- **API определяет поведение библиотеки потоков. Реализация – на усмотрение авторов библиотеки.**
- **Распространены в ОС типа UNIX.**

Потоки в Solaris



Процесс в Solaris



Процесс в Solaris

Потоки в Windows 2000

- **Реализуют схему “один / один”**
- **Каждый поток содержит**

- идентификатор потока (thread id)

- набор регистров

- отдельные стеки для пользовательских и системных процедур

- область памяти для локальных данных потока (TLS – thread-local storage)

Потоки в Linux

- **В Linux потоки называются *tasks (задачами)*, а не *threads*.**
- **Поток создается системным вызовом clone().**
- **clone() полволяет дочерней задаче использовать общее адресное пространство с родительской задачей (процессом)**

Потоки в Java

- **Потоки в Java могут быть созданы следующими способами:**
  - **Как расширения класса Thread**
  - **Как классы, реализующие интерфейс Runnable**
- **Потоки в Java управляются JVM**
- **Возможно создание групп потоков и иерархии таких групп**

Состояния потоков в Java