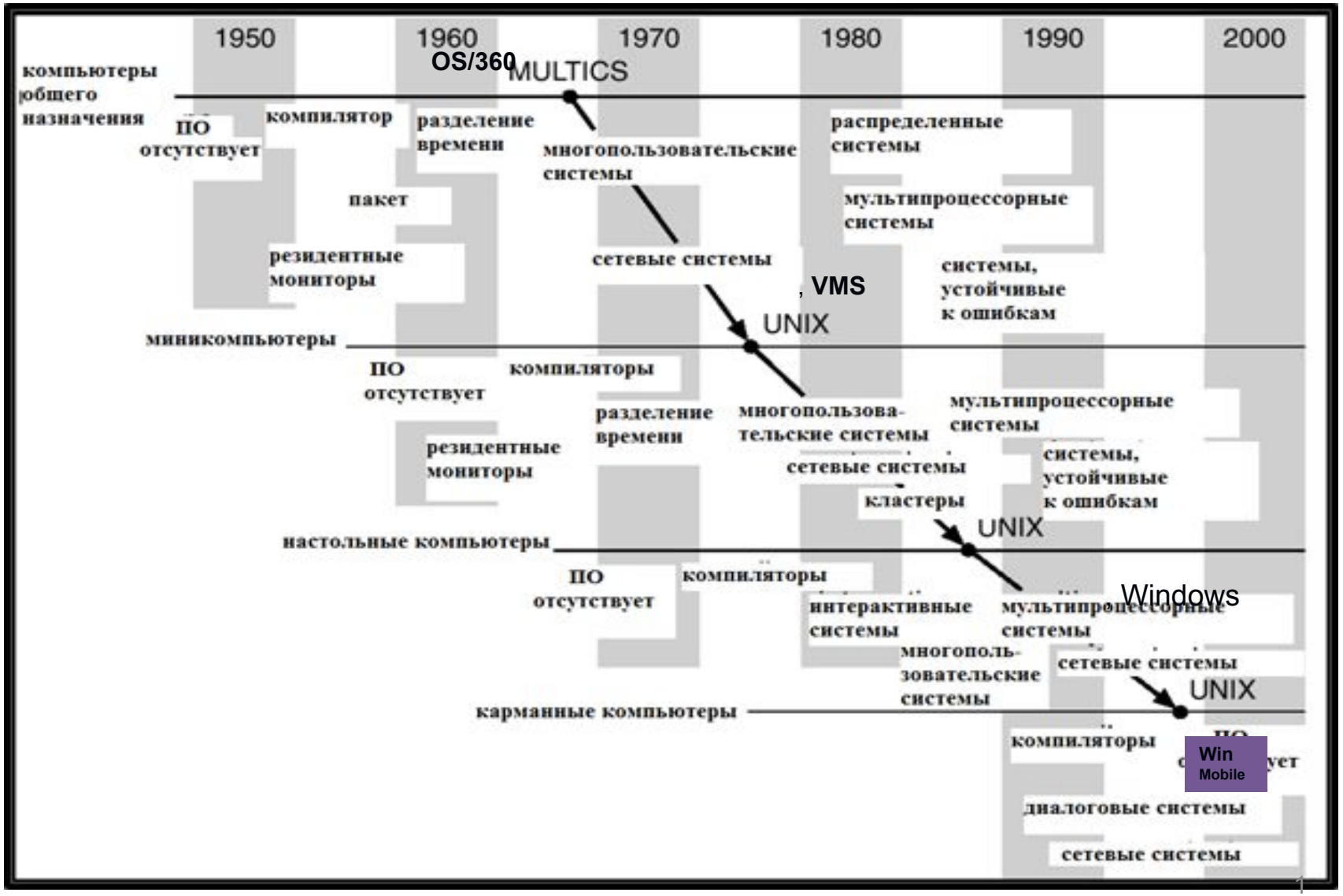
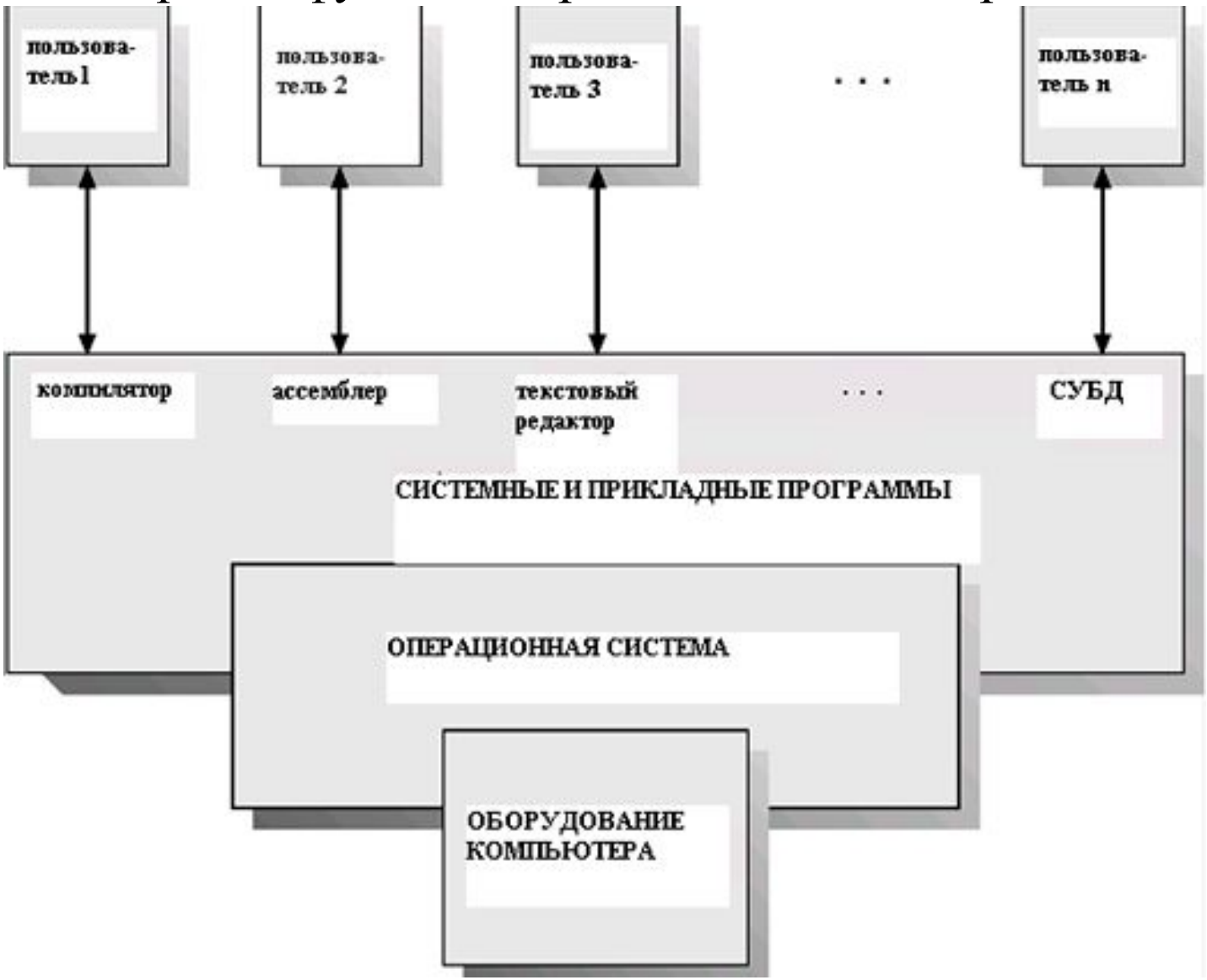
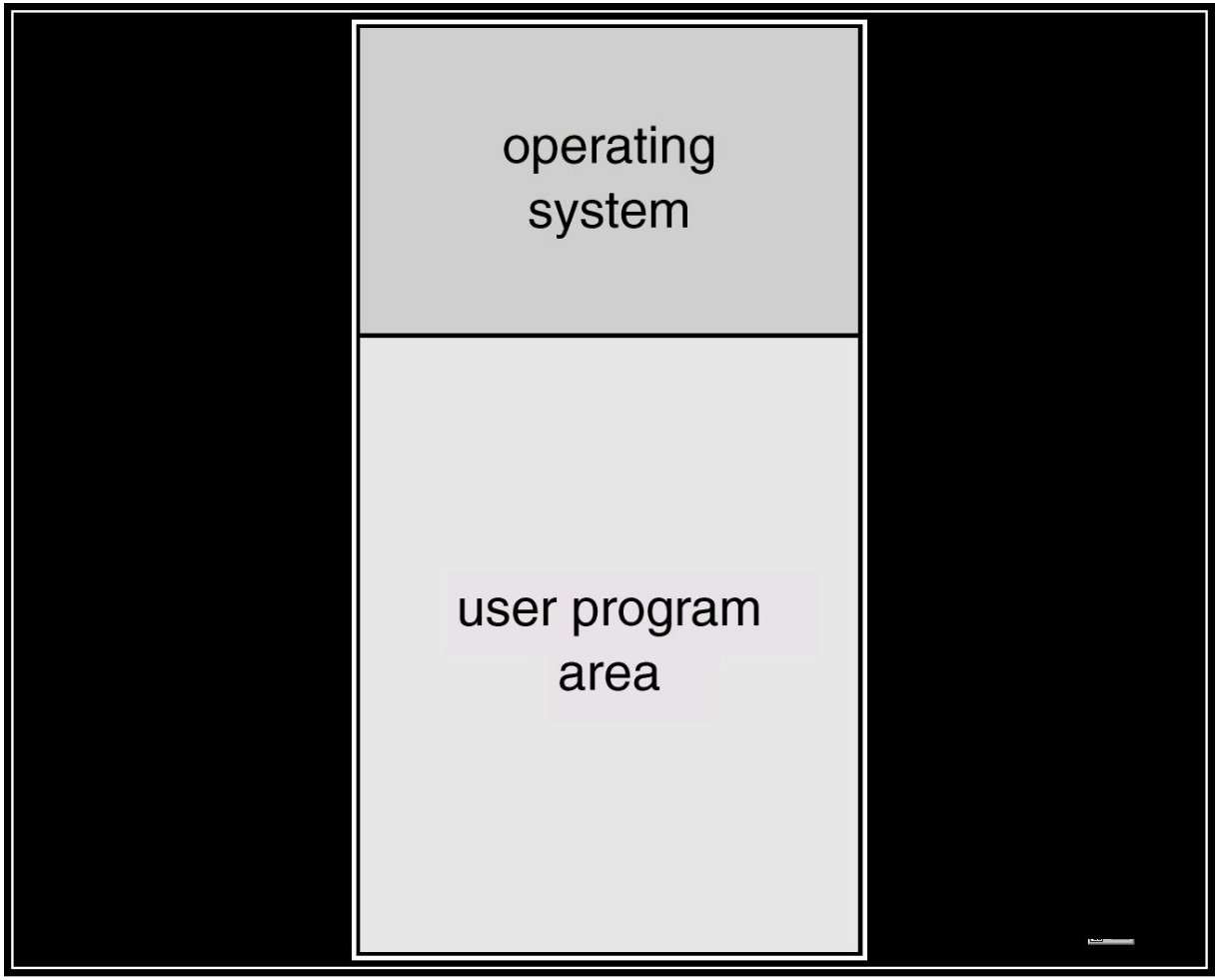
Развитие концепций и возможностей ОС



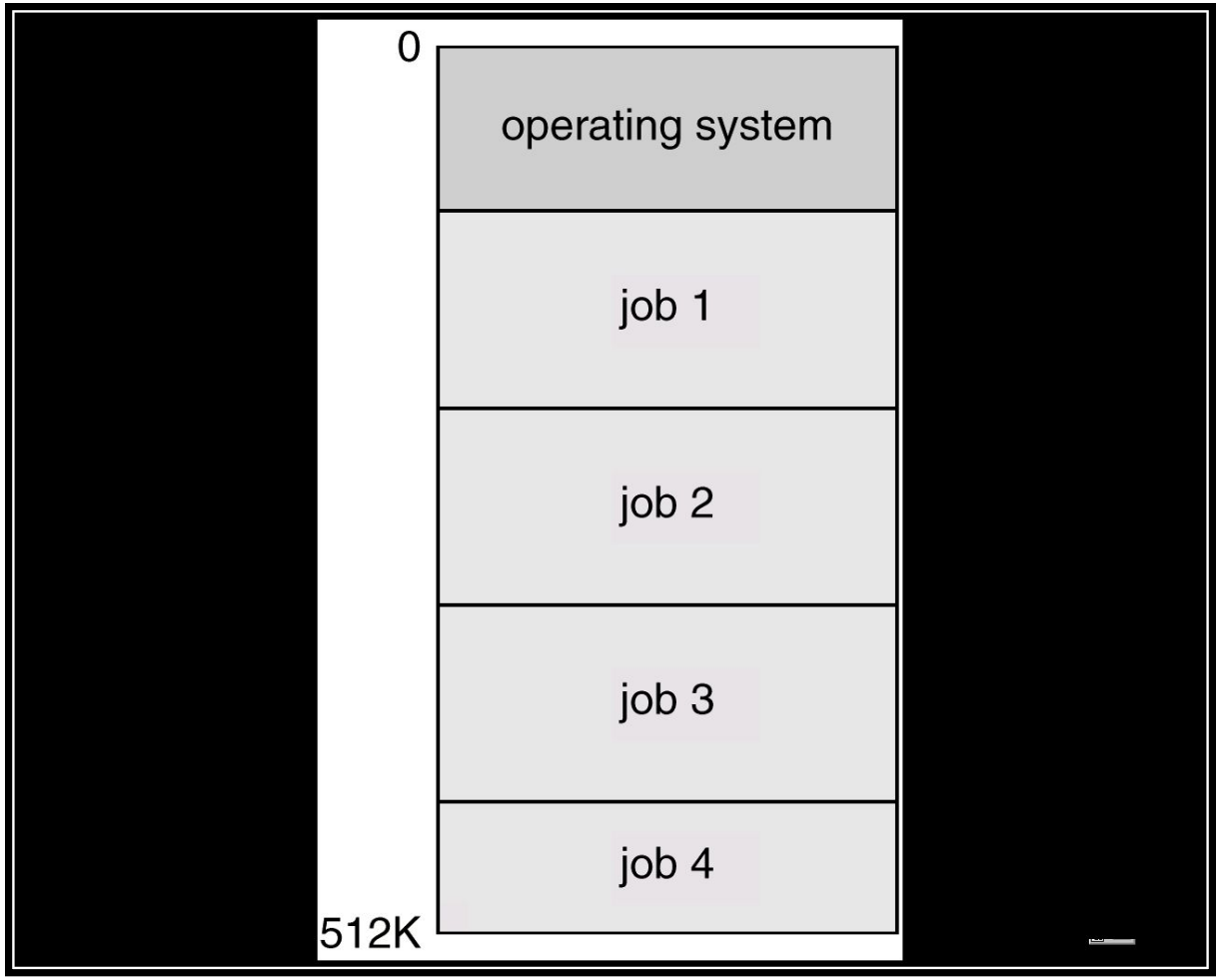
Общая картина функционирования компьютерной системы



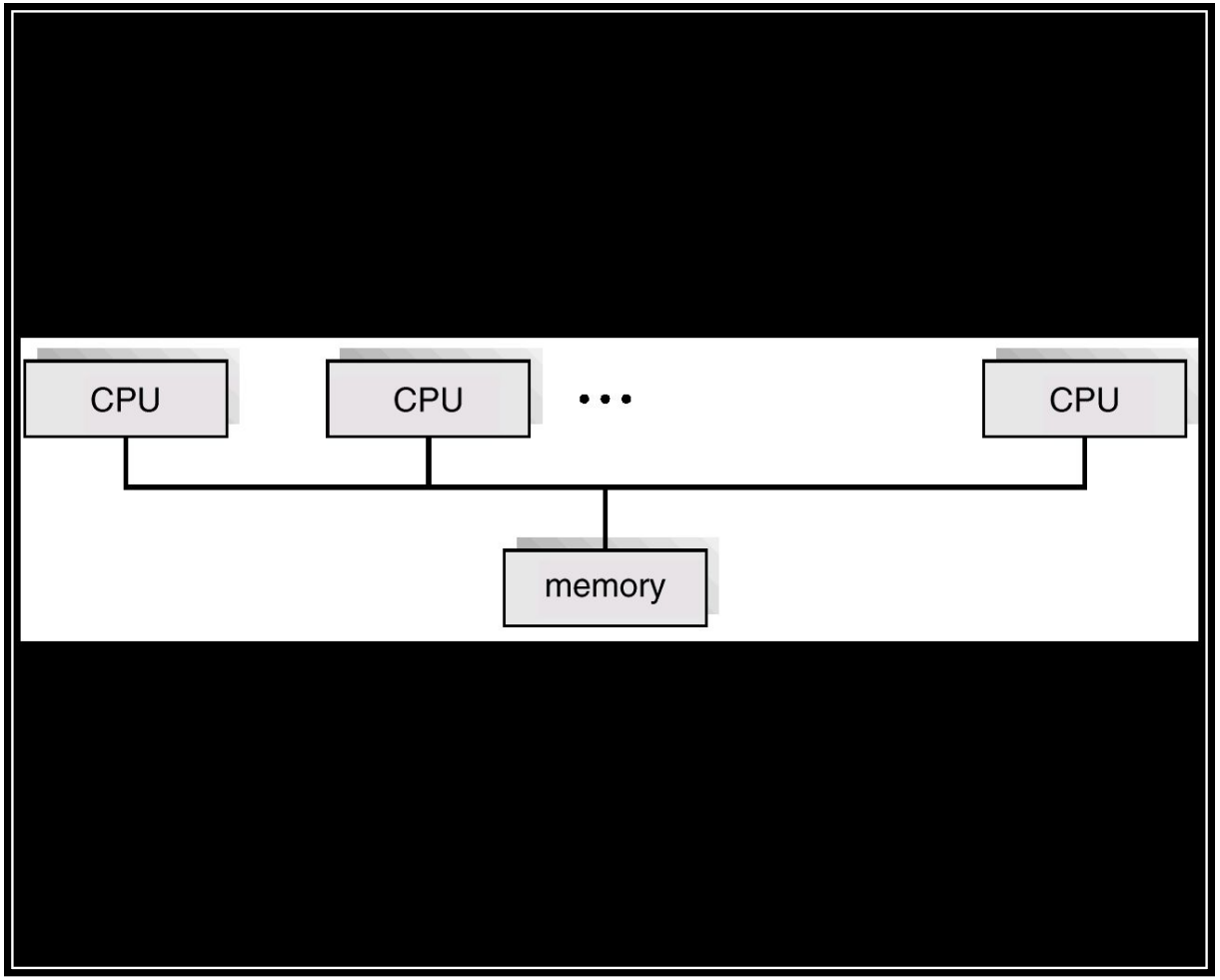
Распределение памяти в простой системе пакетной обработки



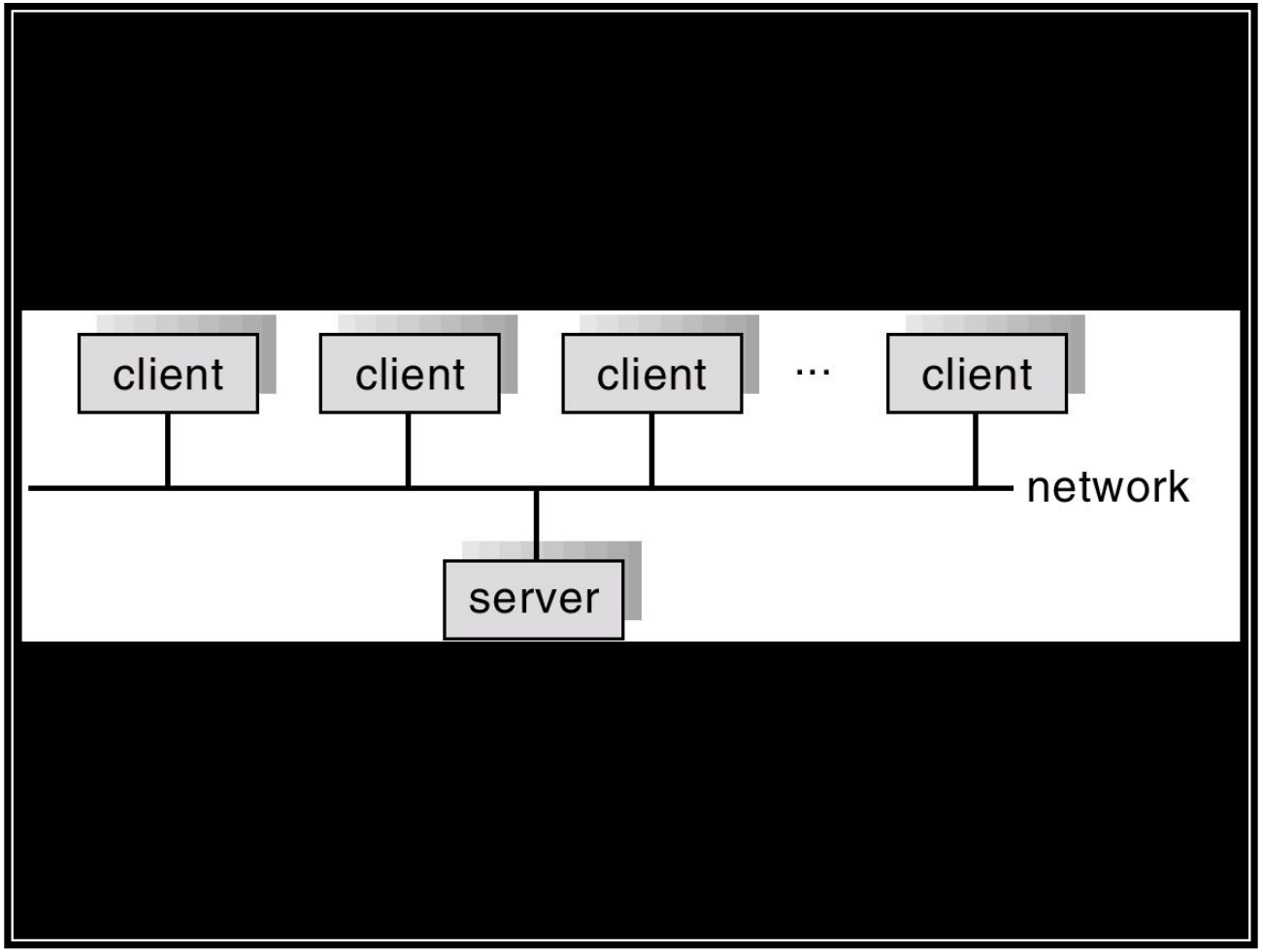
Системы пакетной обработки с поддержкой мультипрограммирования



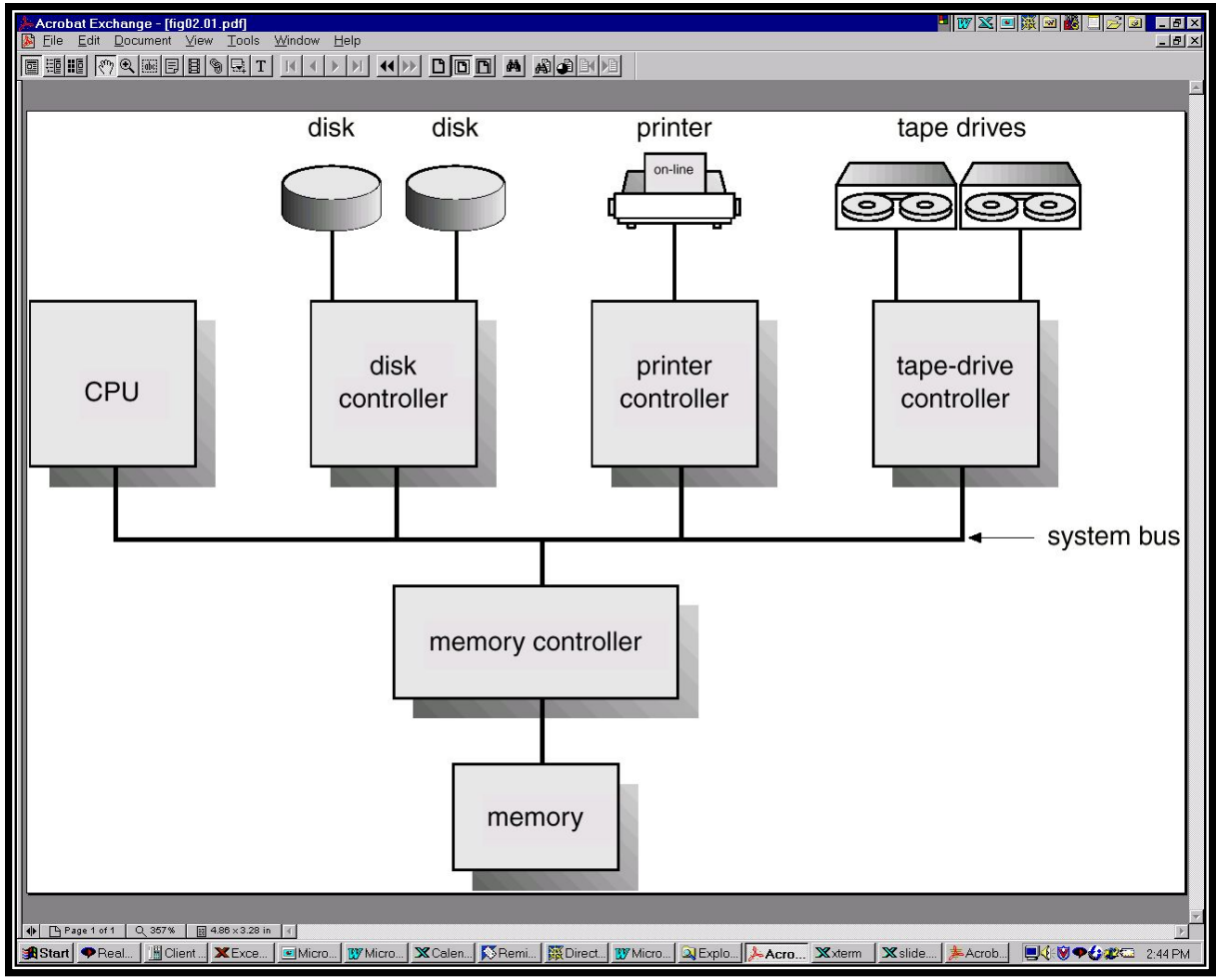
SMP-архитектура



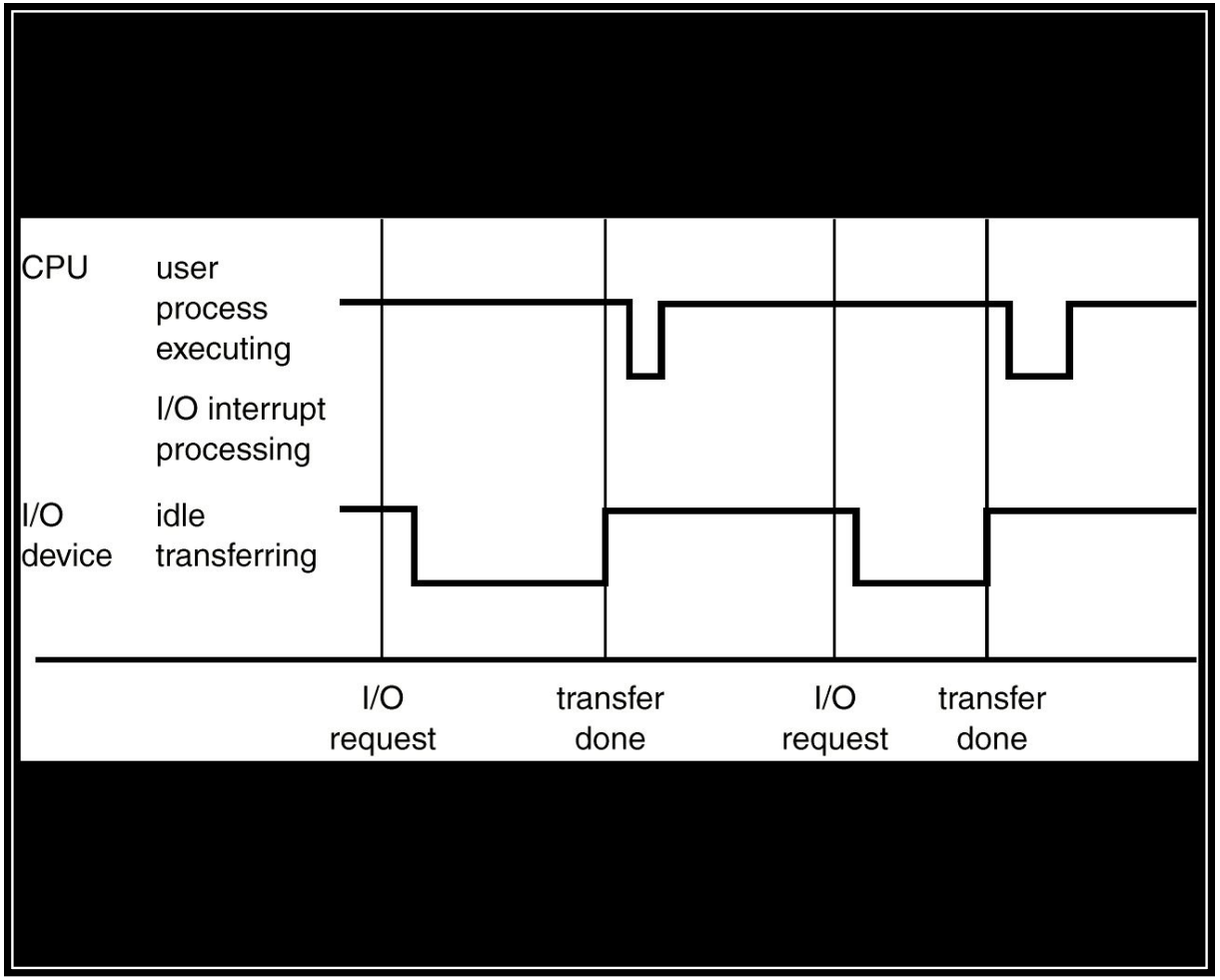
Общая структура клиент-серверной системы



Архитектура компьютерных систем 2/2



Временной график прерываний процесса, выполняющего вывод



Два метода ввода-вывода: синхронный и асинхронный

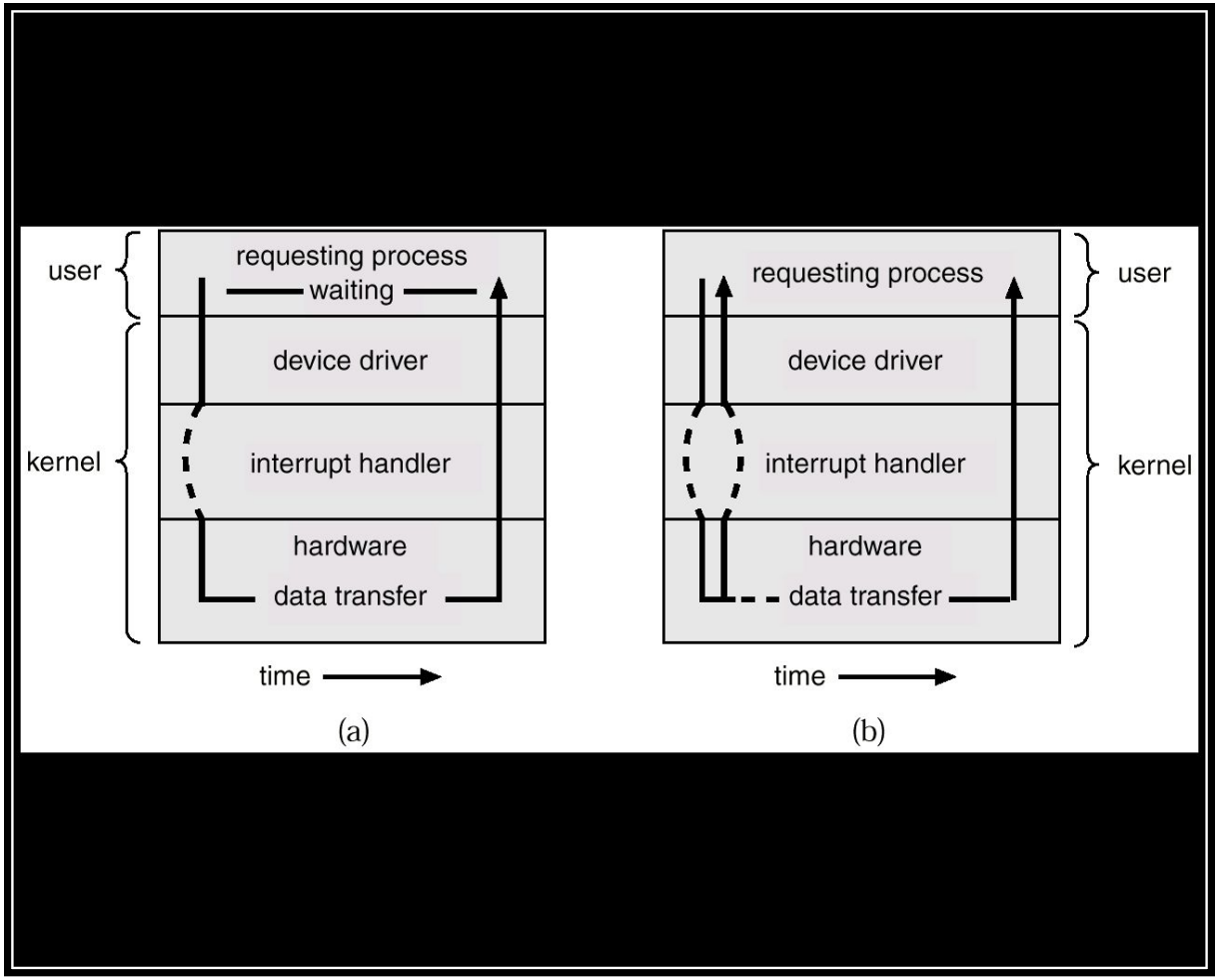
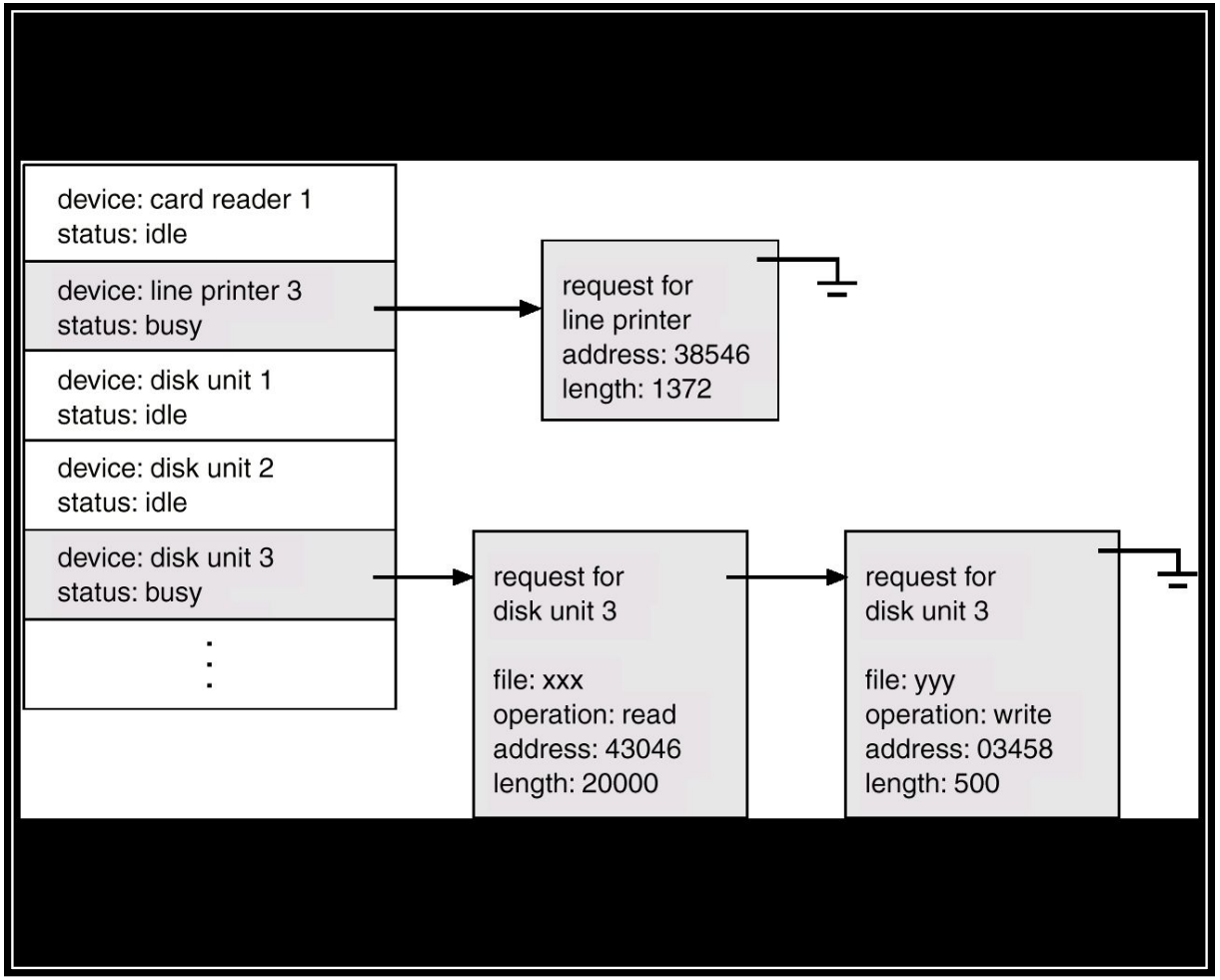
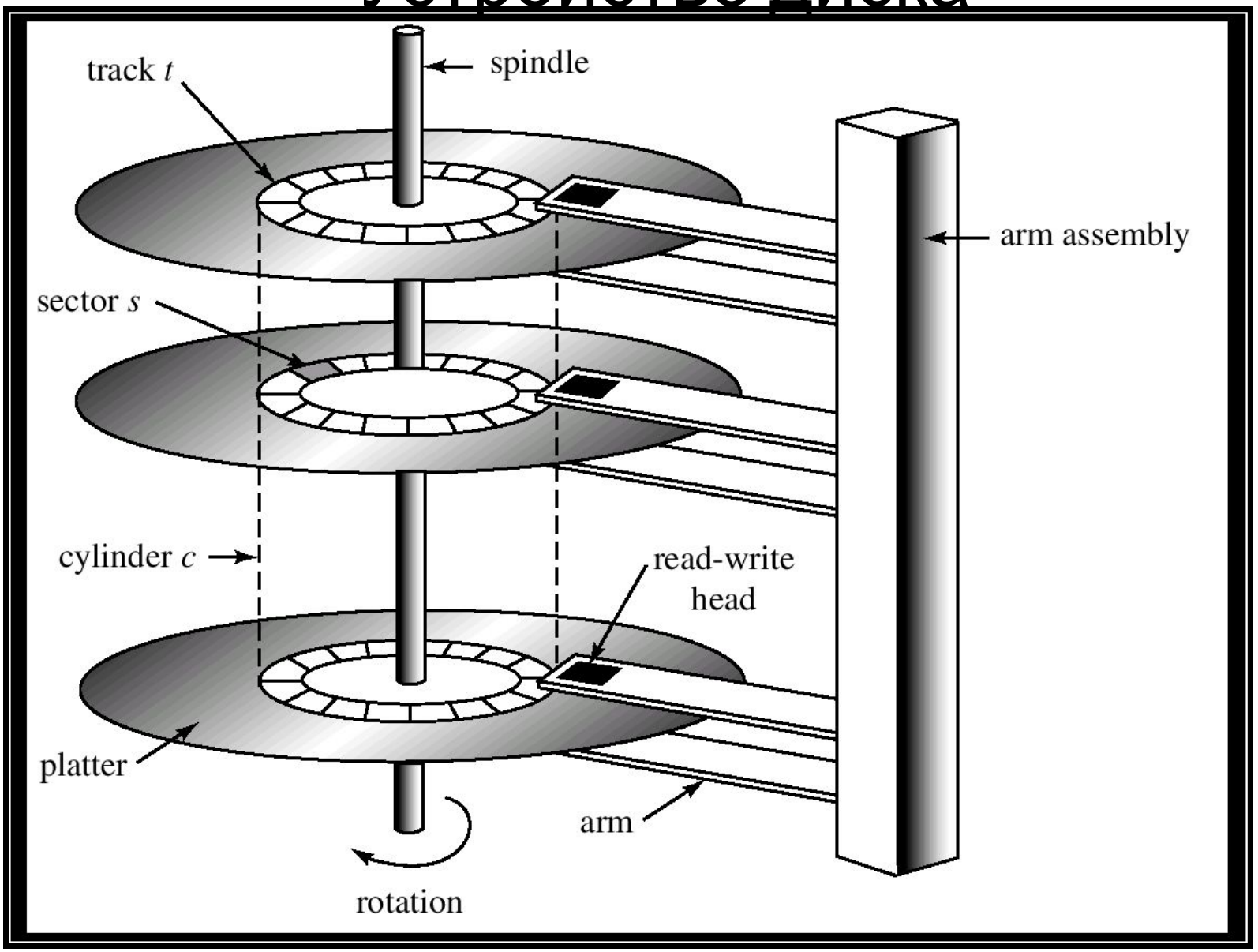


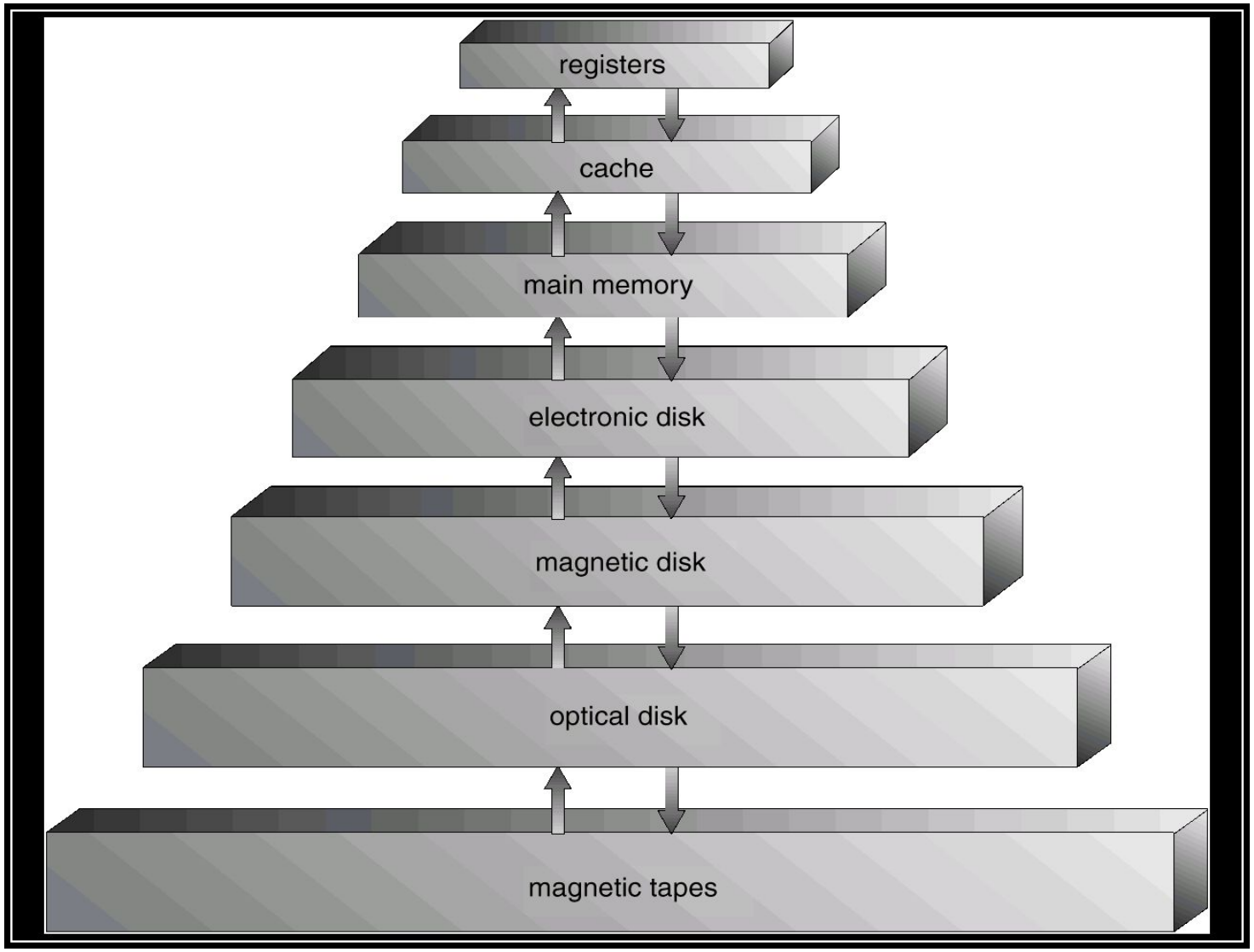
Таблица состояния устройств



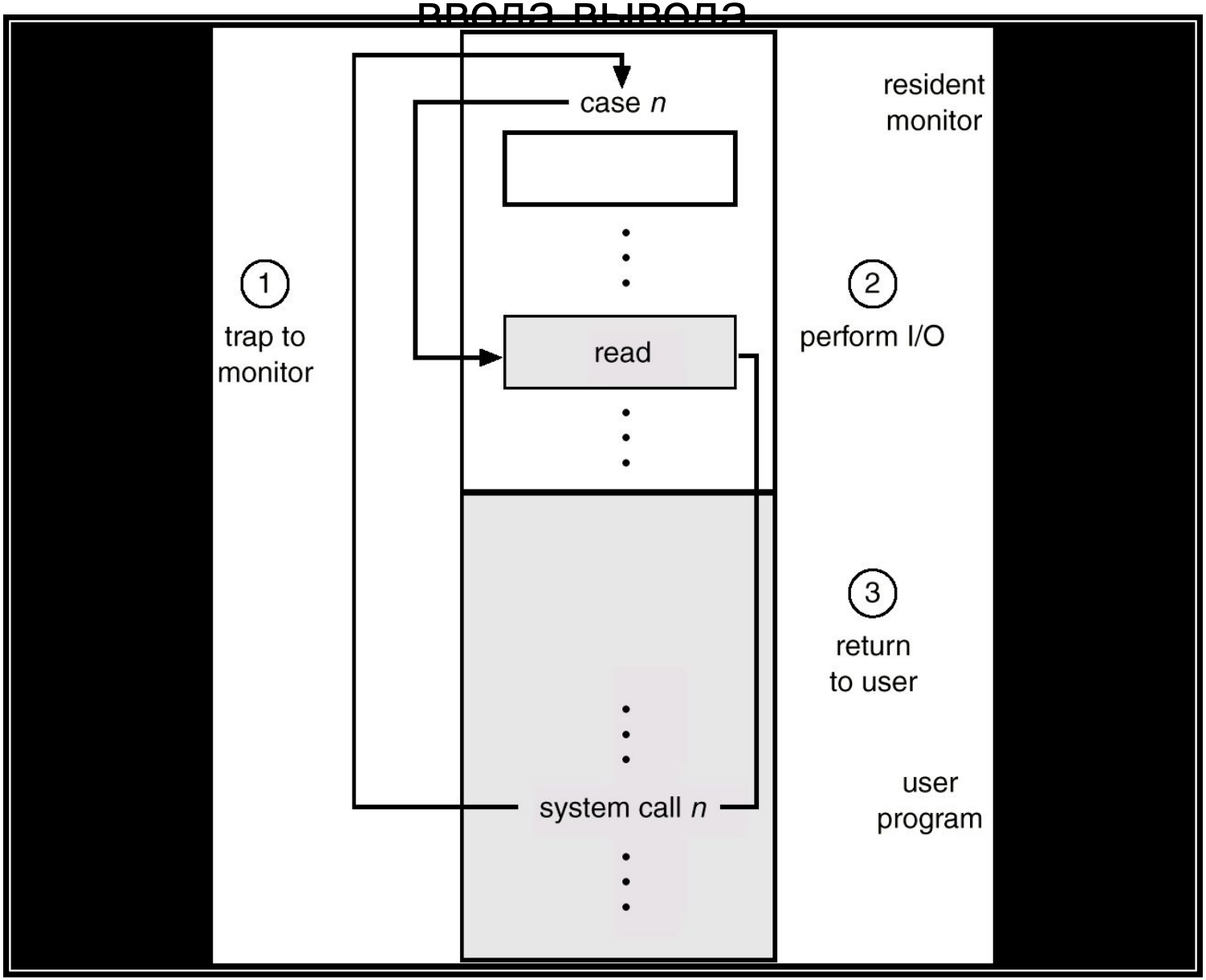
Устройство диска



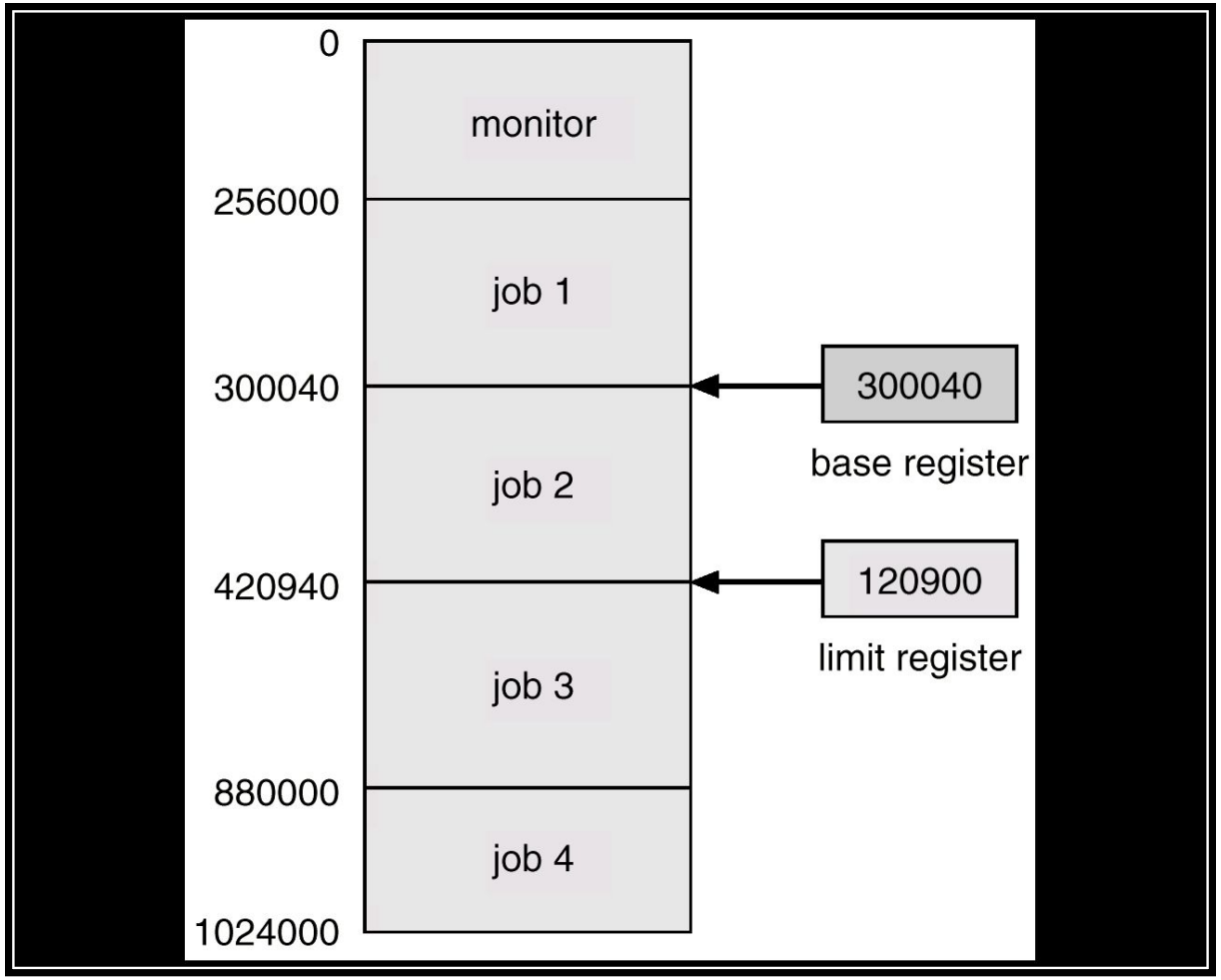
Иерархия устройств памяти



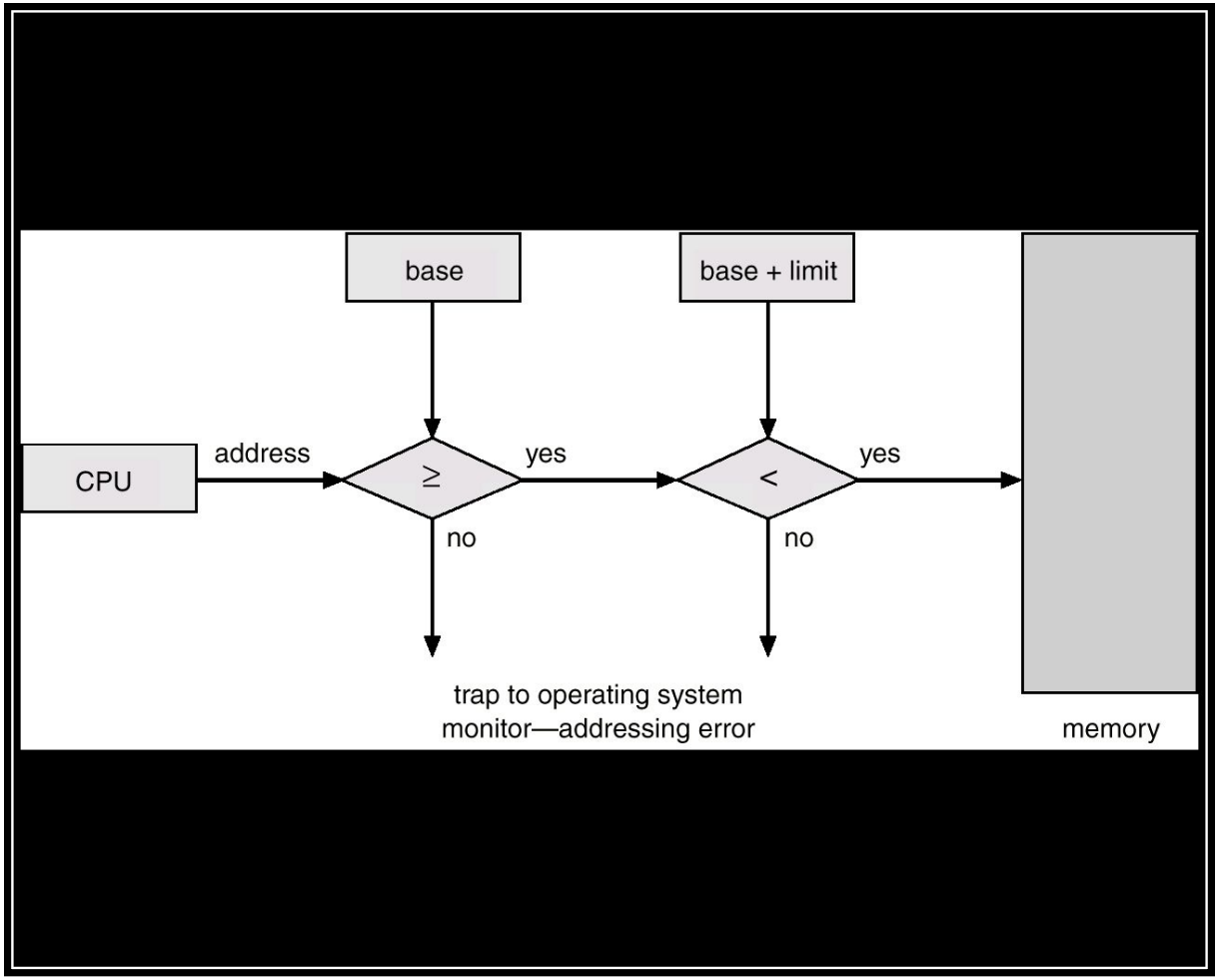
Использование системного вызова для выполнения ввода-вывода



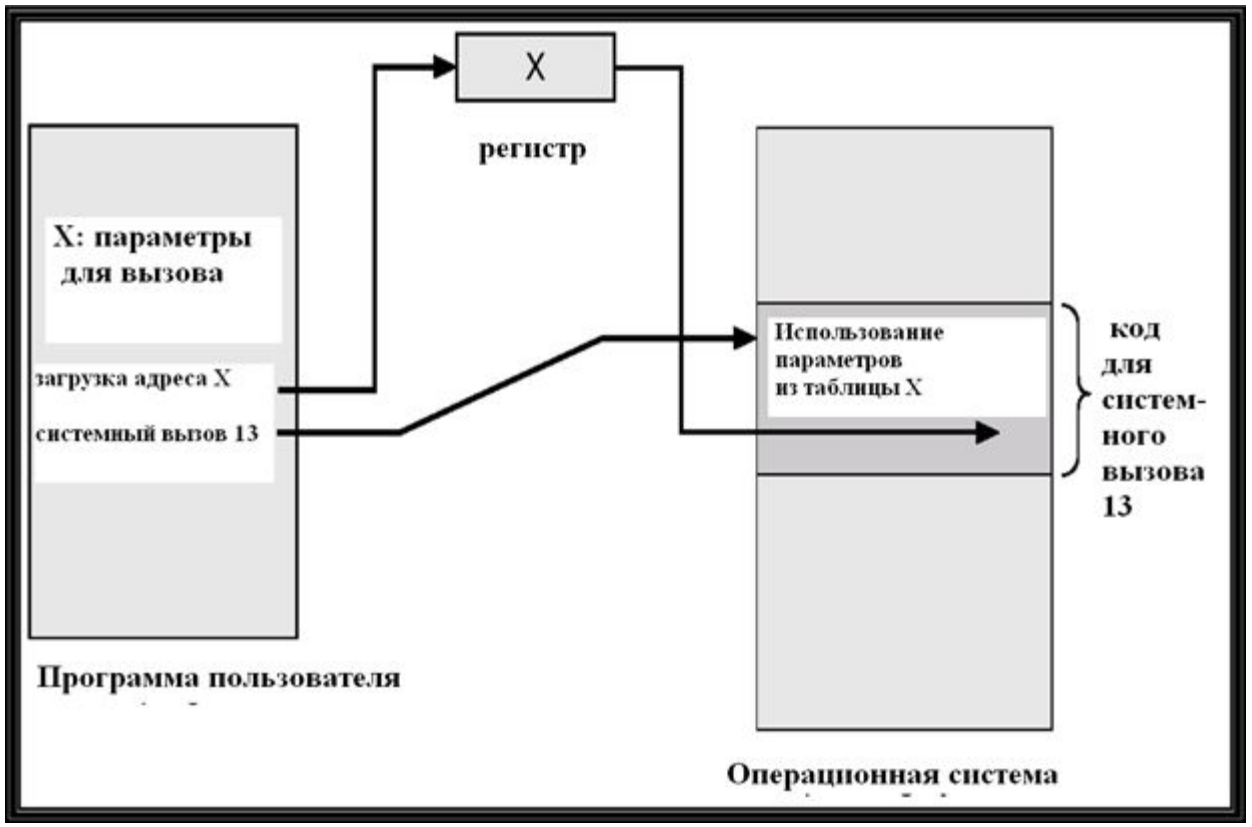
Использование базового регистра и регистра границы



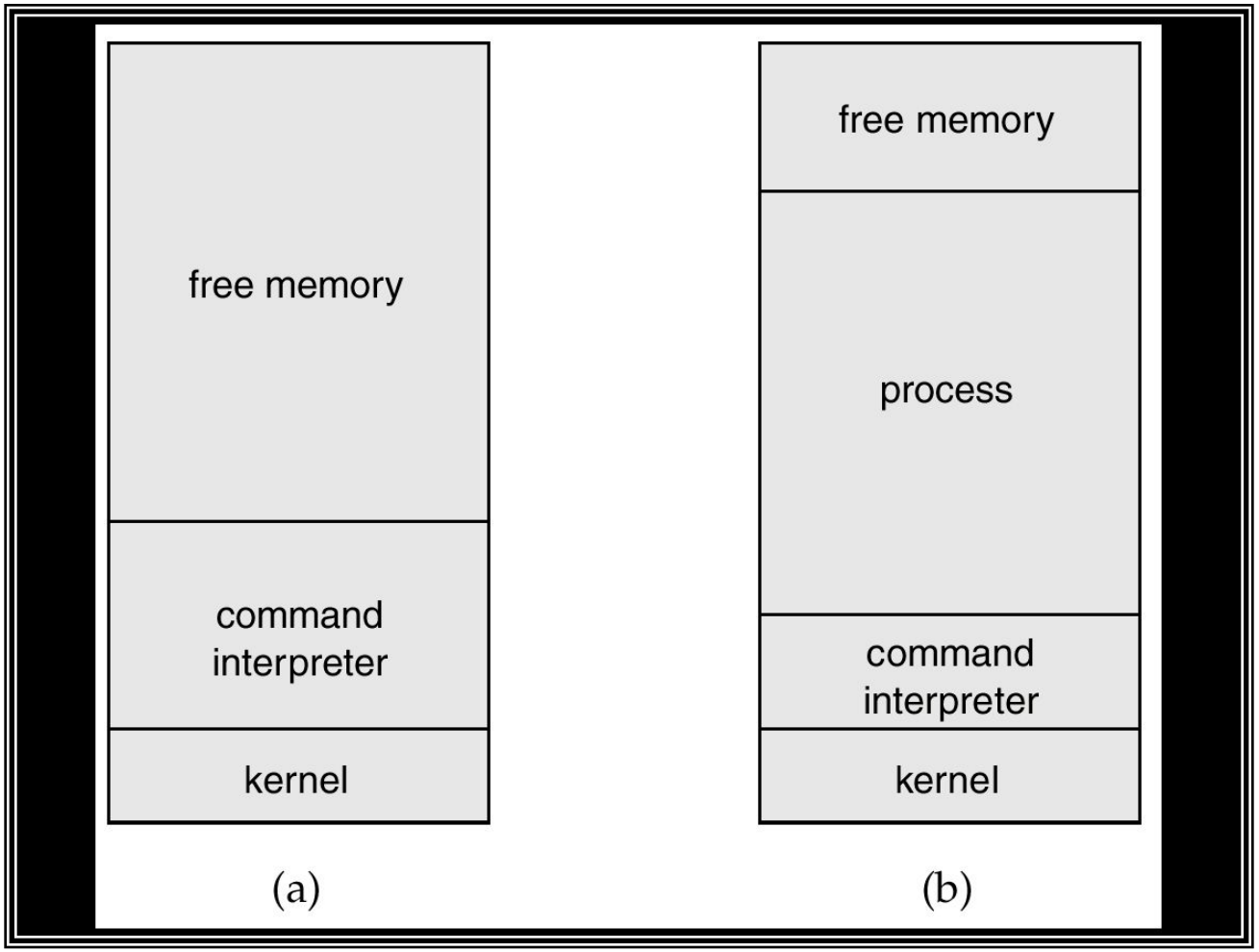
Аппаратная защита адресов памяти



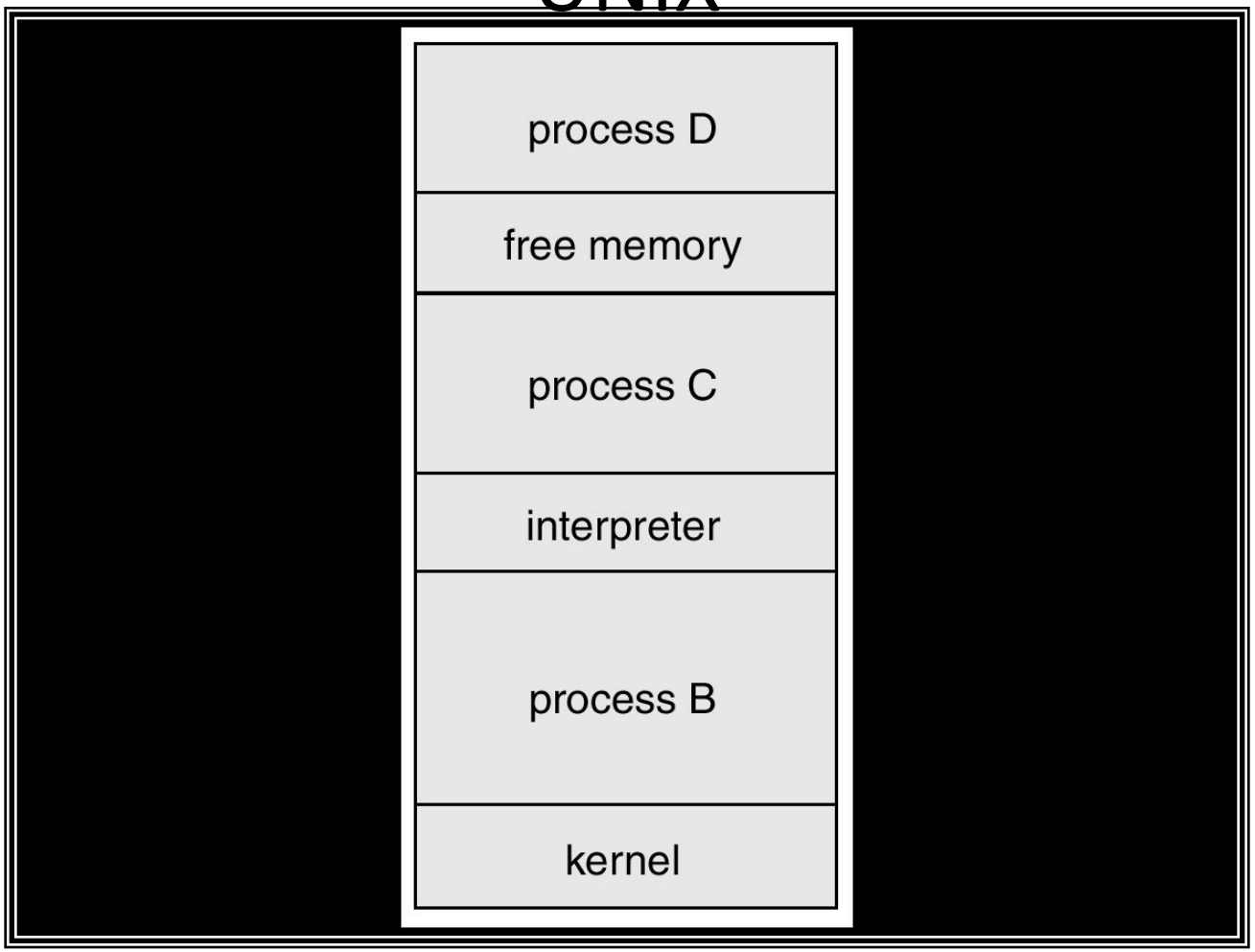
Передача параметров в таблице



Исполнение программ в MS-DOS

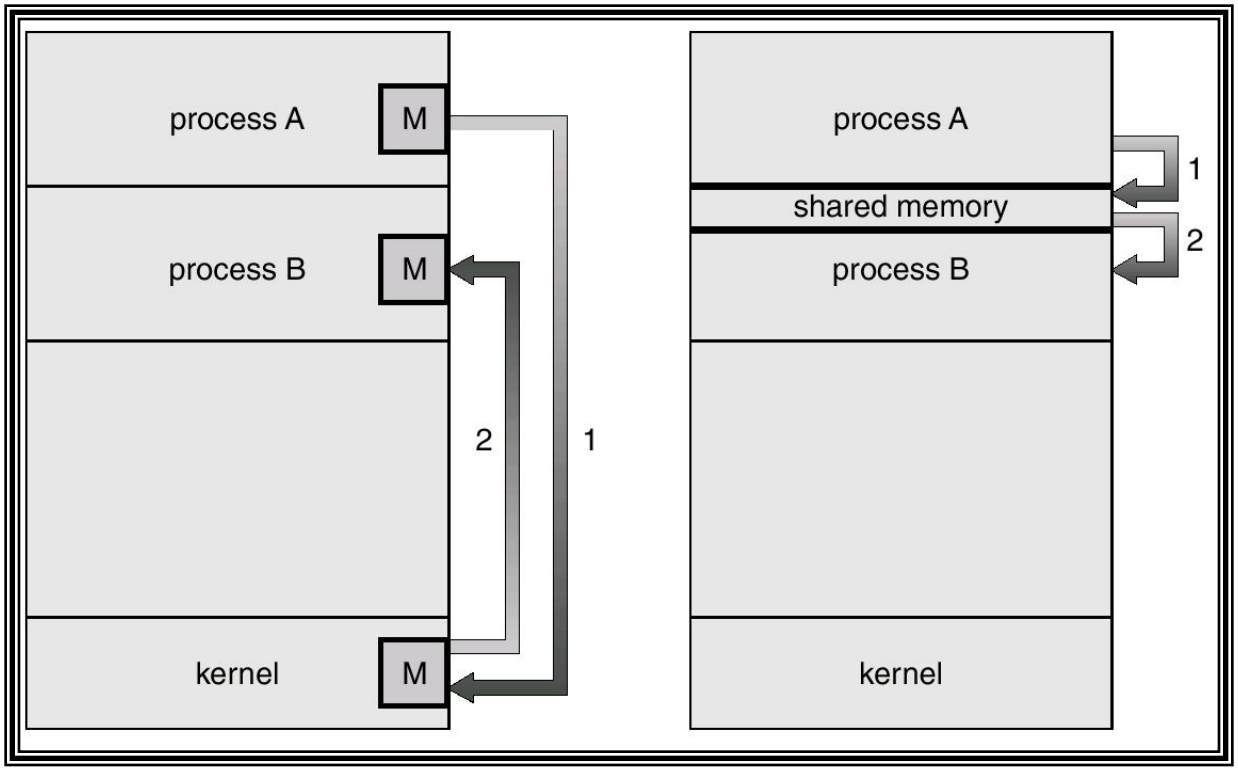


Исполнение нескольких программ в UNIX

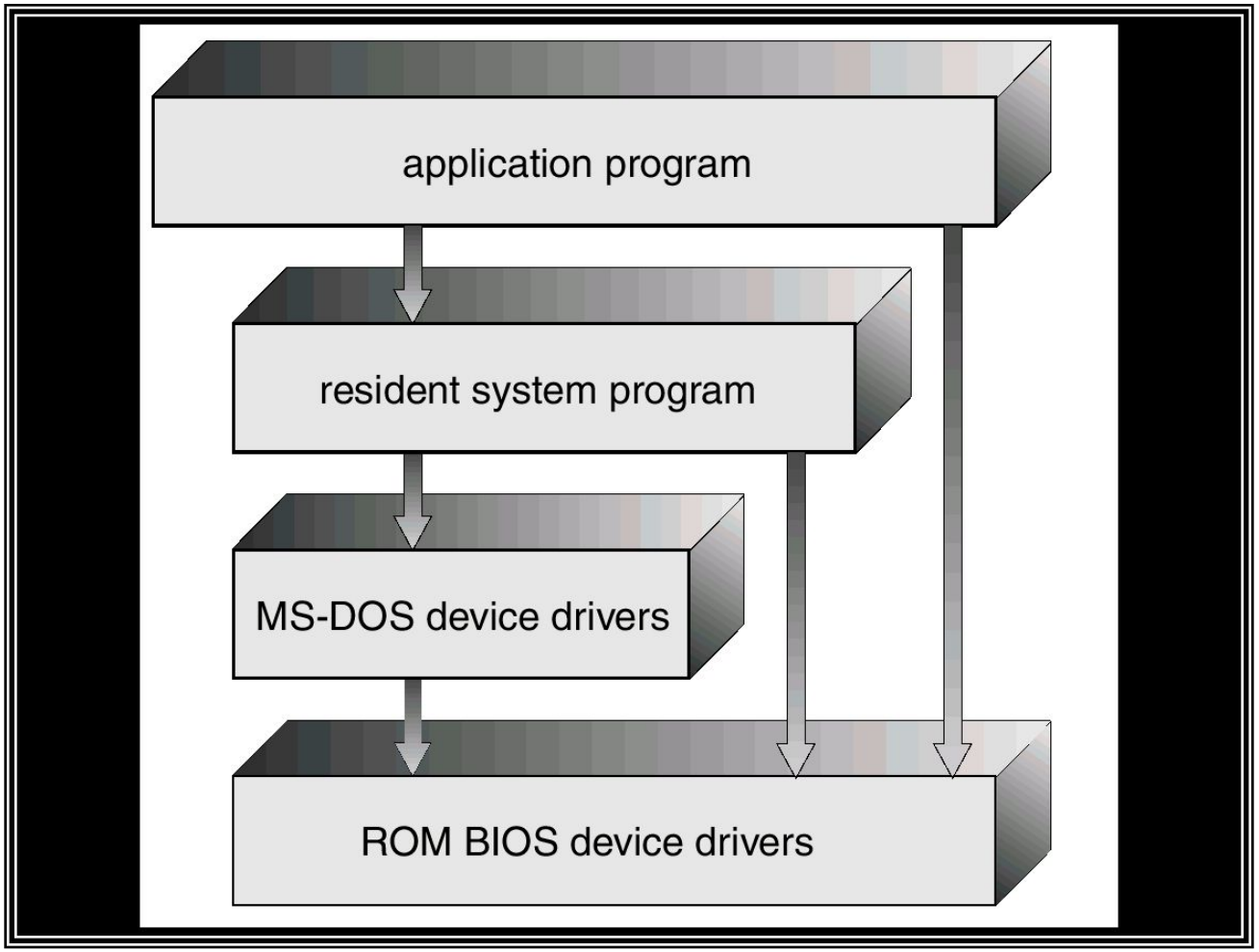


Коммуникационные модели

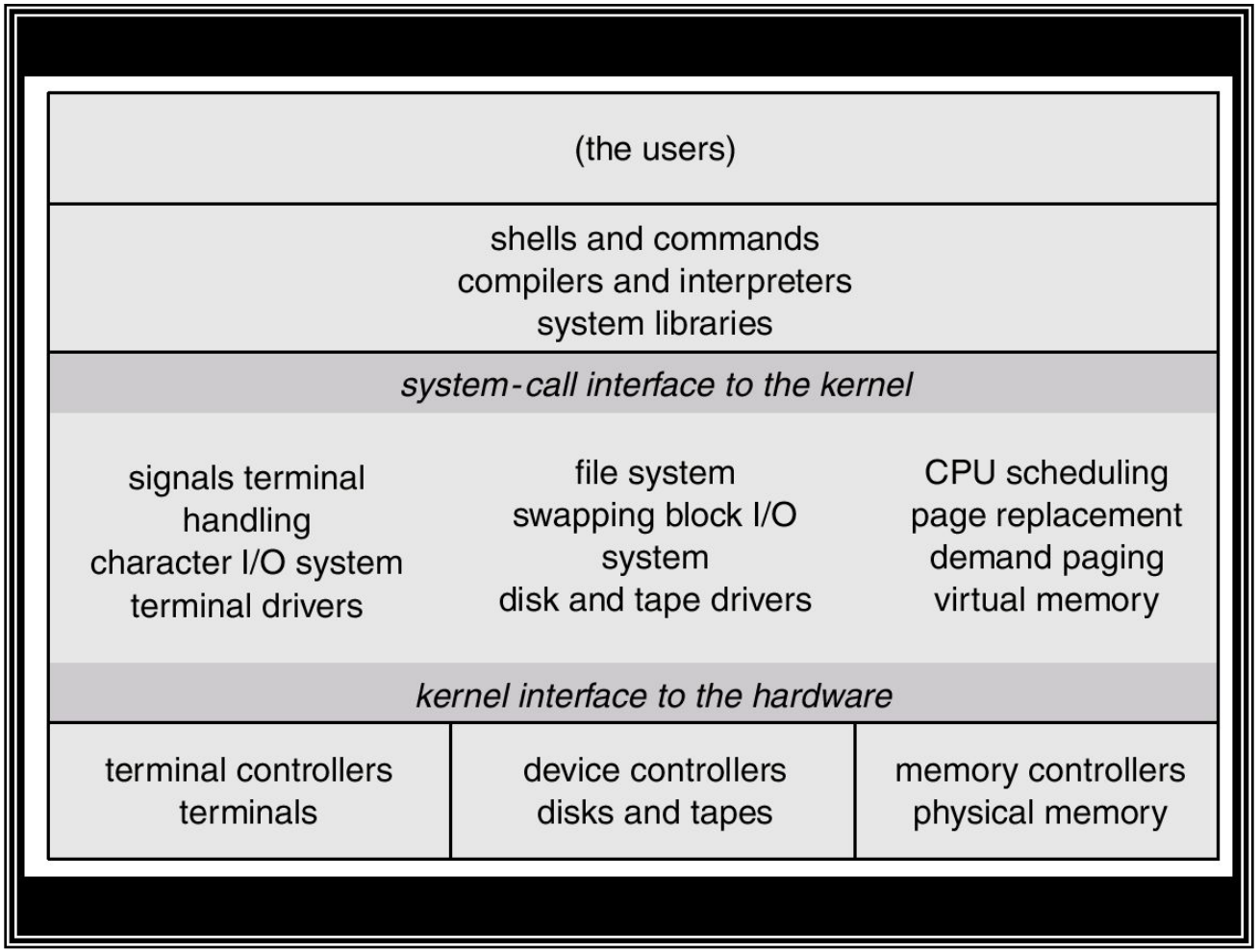
- **Могут реализовываться с помощью общей памяти или передачи сообщений**



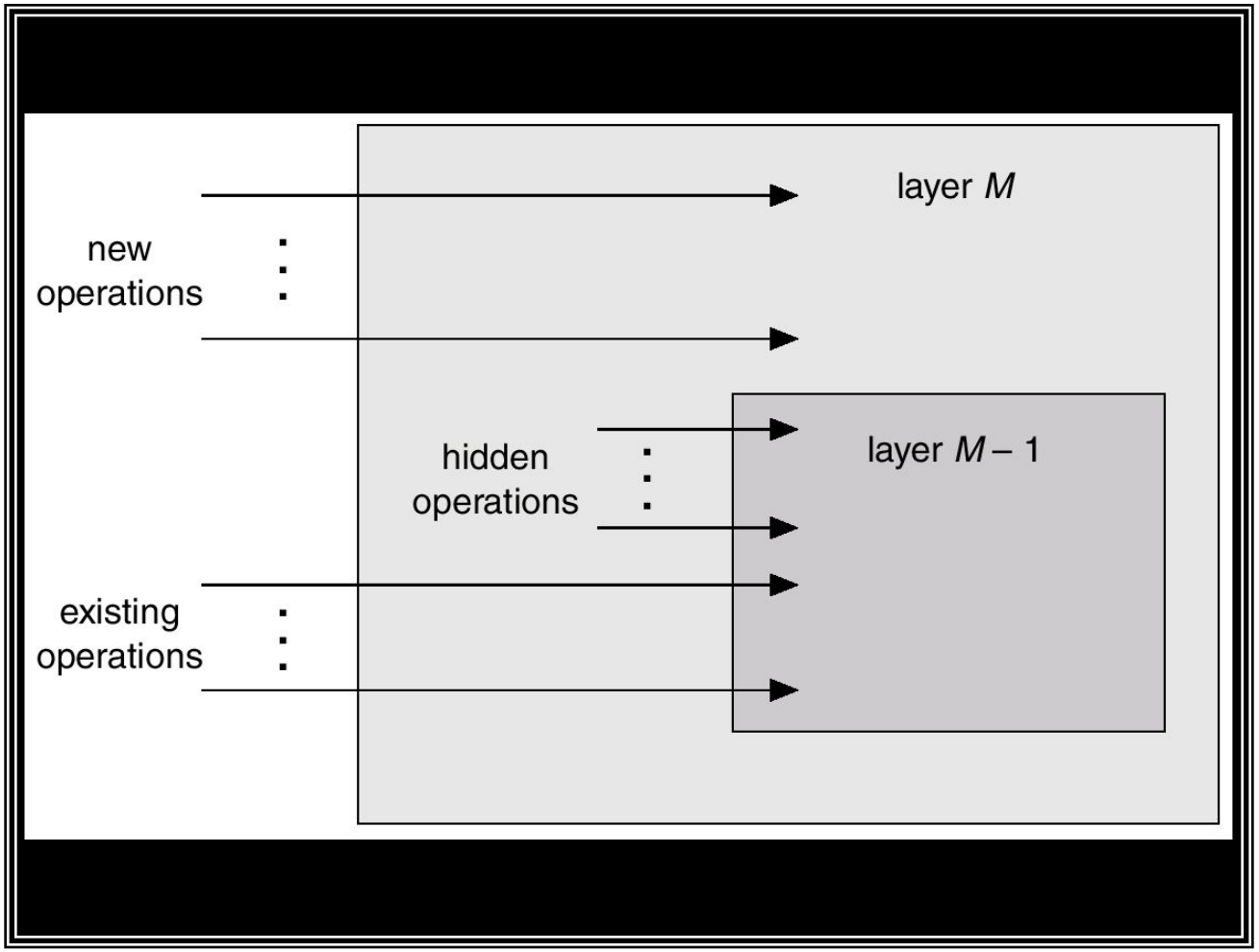
Уровни (абстракции) модулей MS-DOS



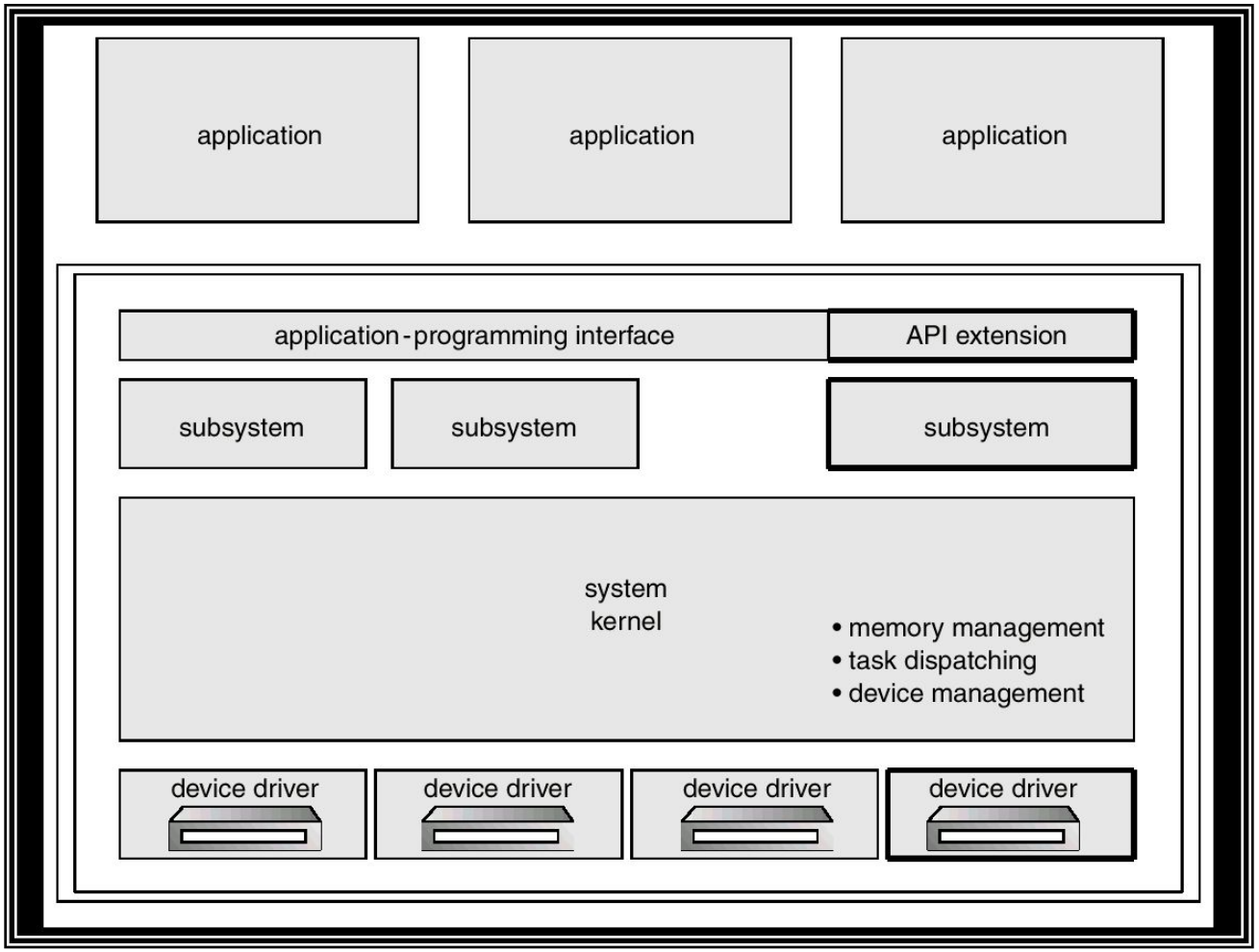
Структура системы UNIX



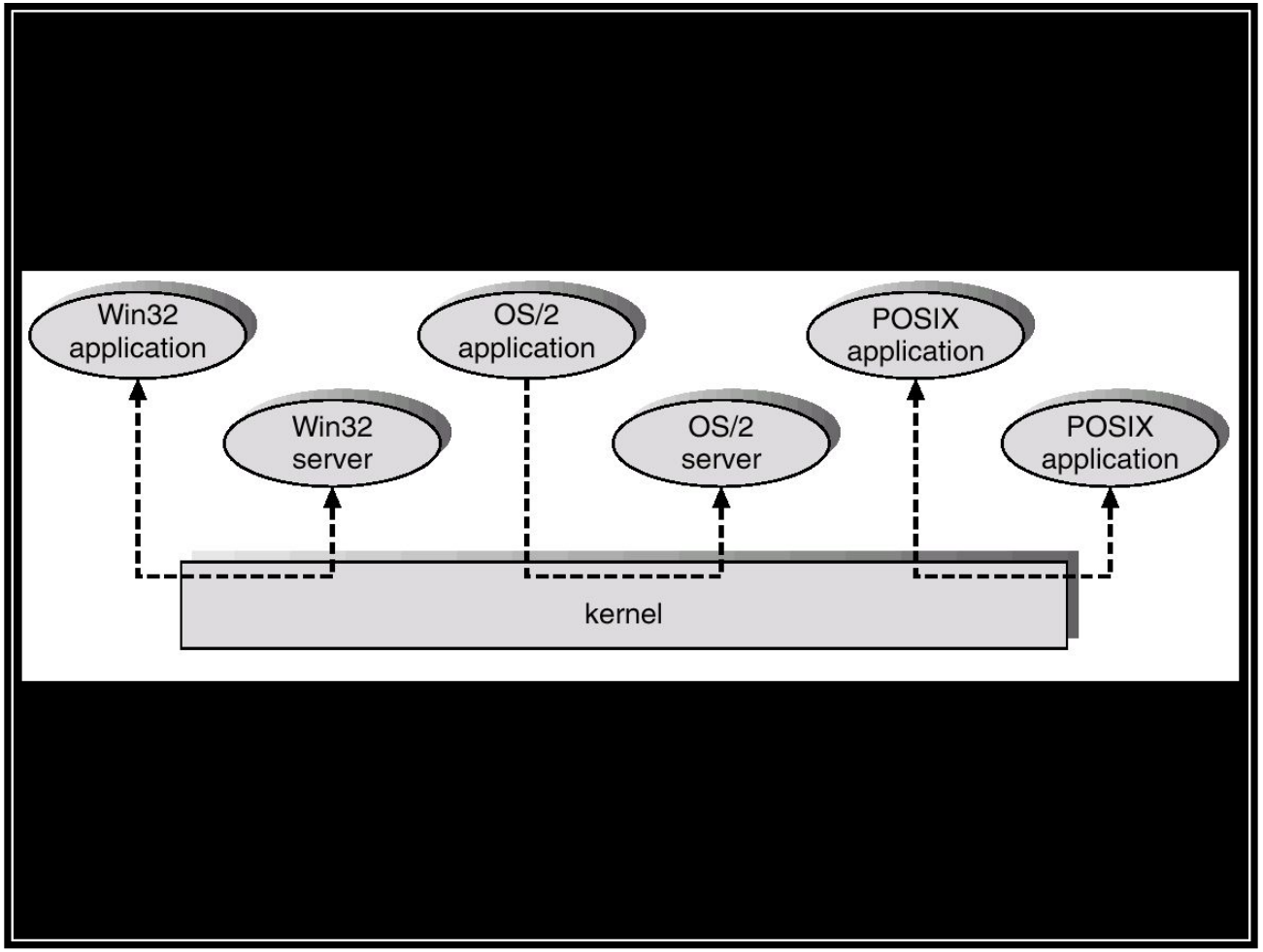
Уровни абстракции ОС



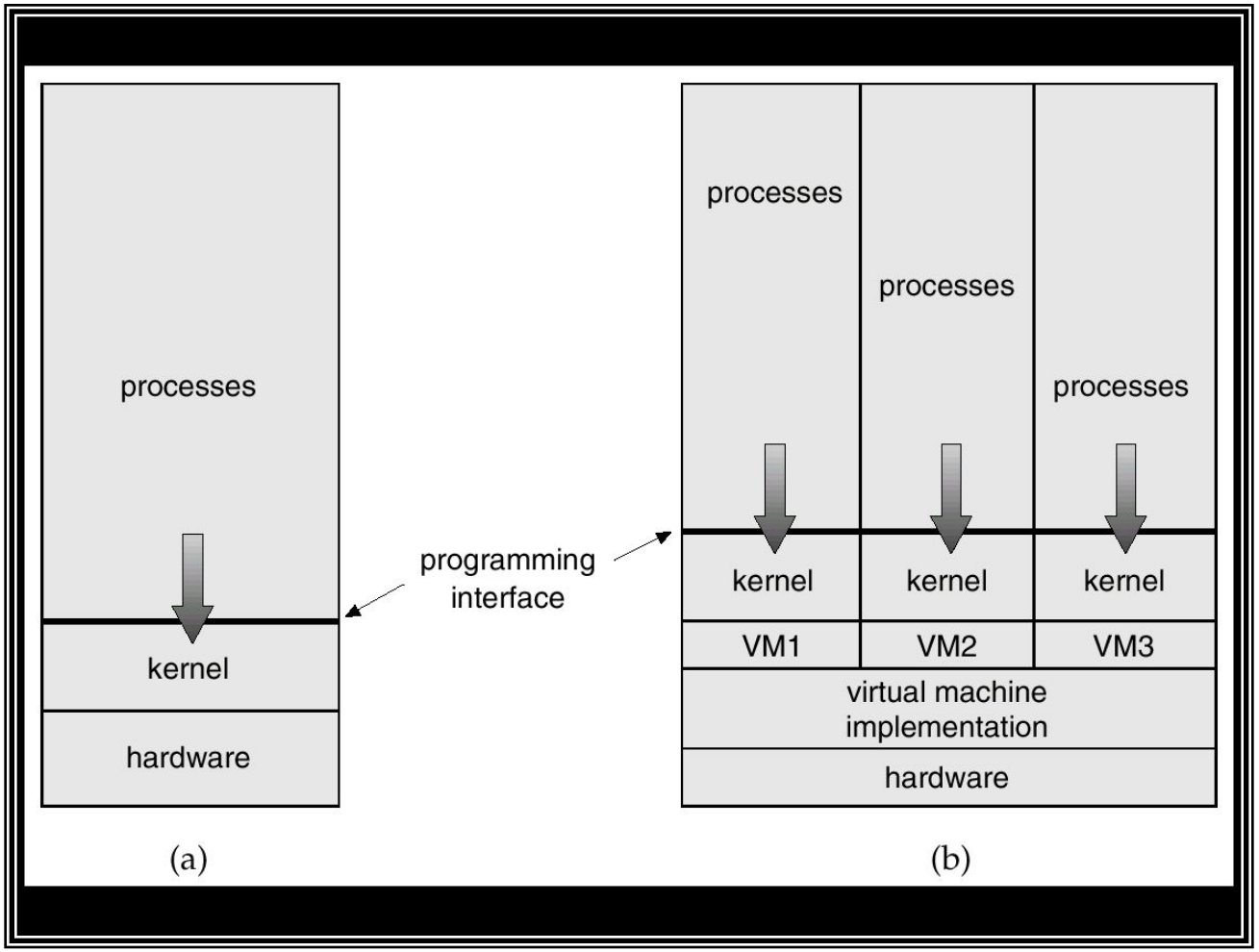
Структура уровней абстракции OS/2



Клиент-серверная структура Windows NT



Модели ОС без использования виртуальных машин и на основе виртуальных машин



Виртуальная машина Java

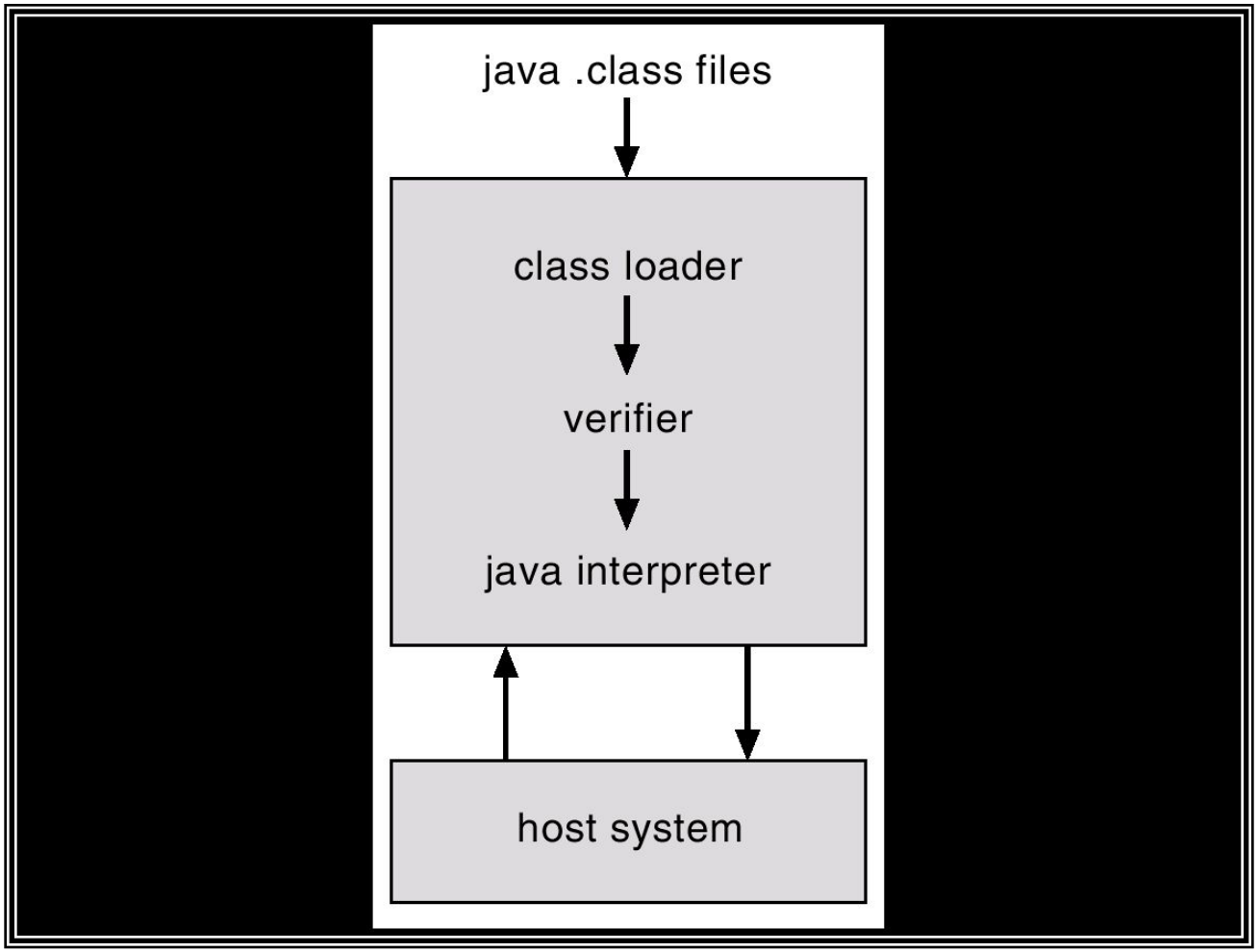
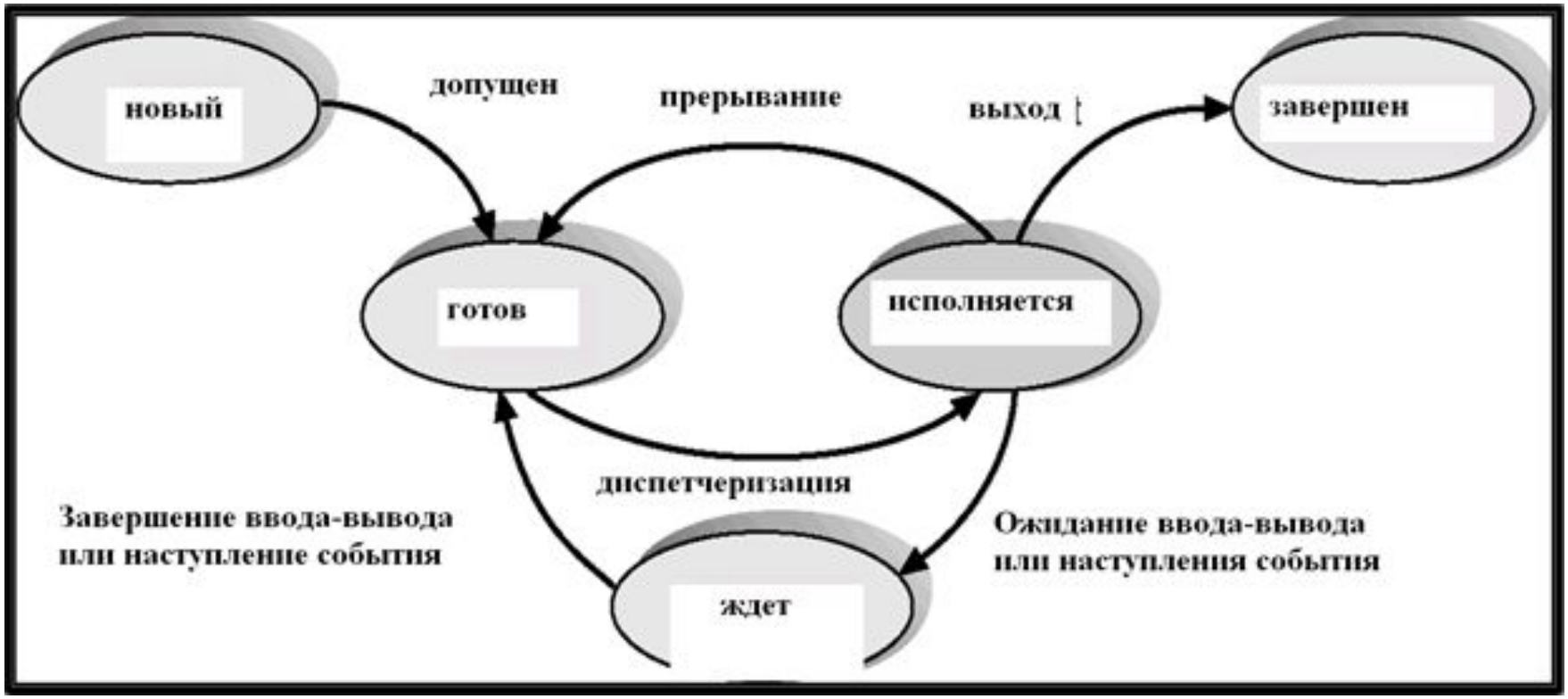
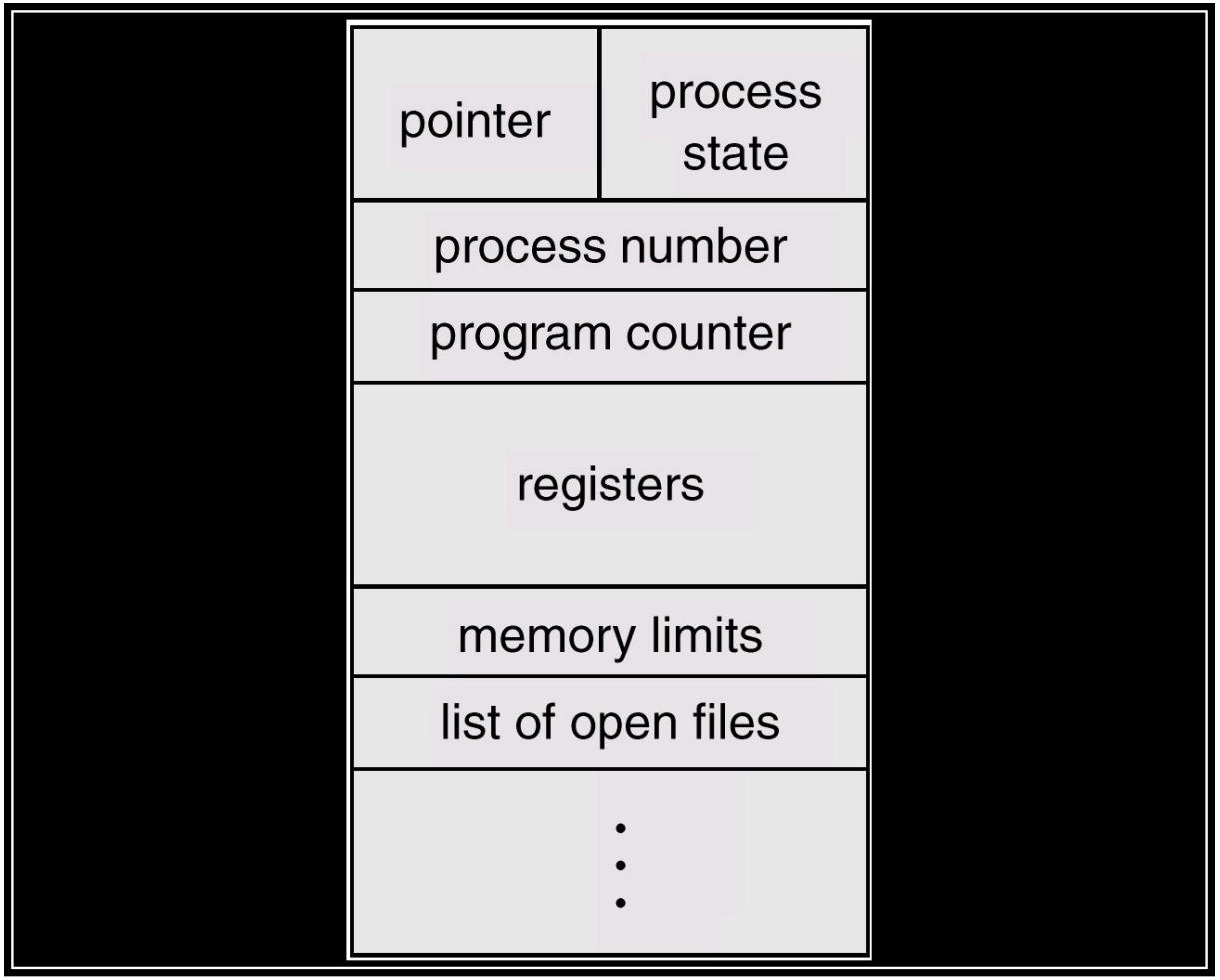


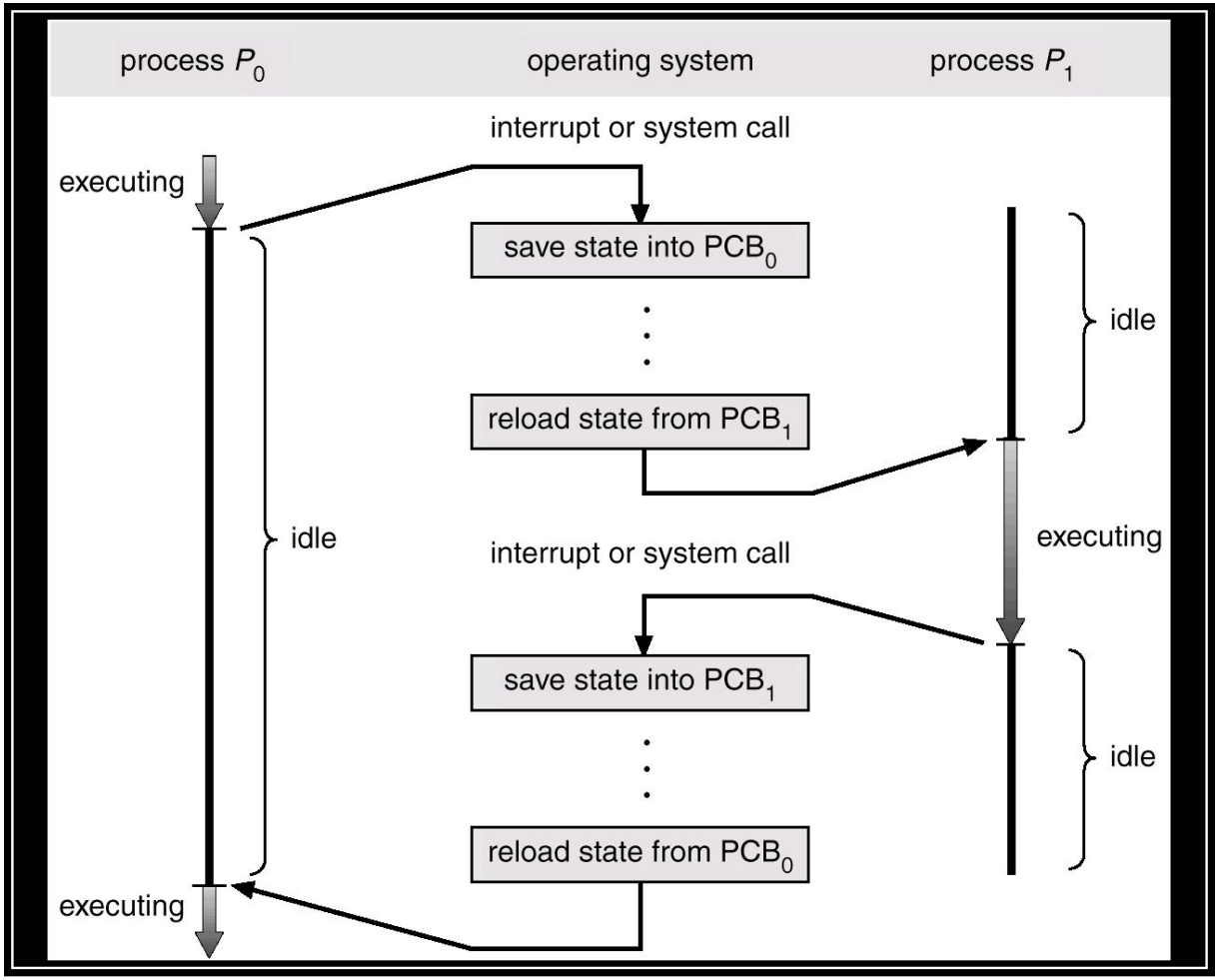
Диаграмма состояний процесса



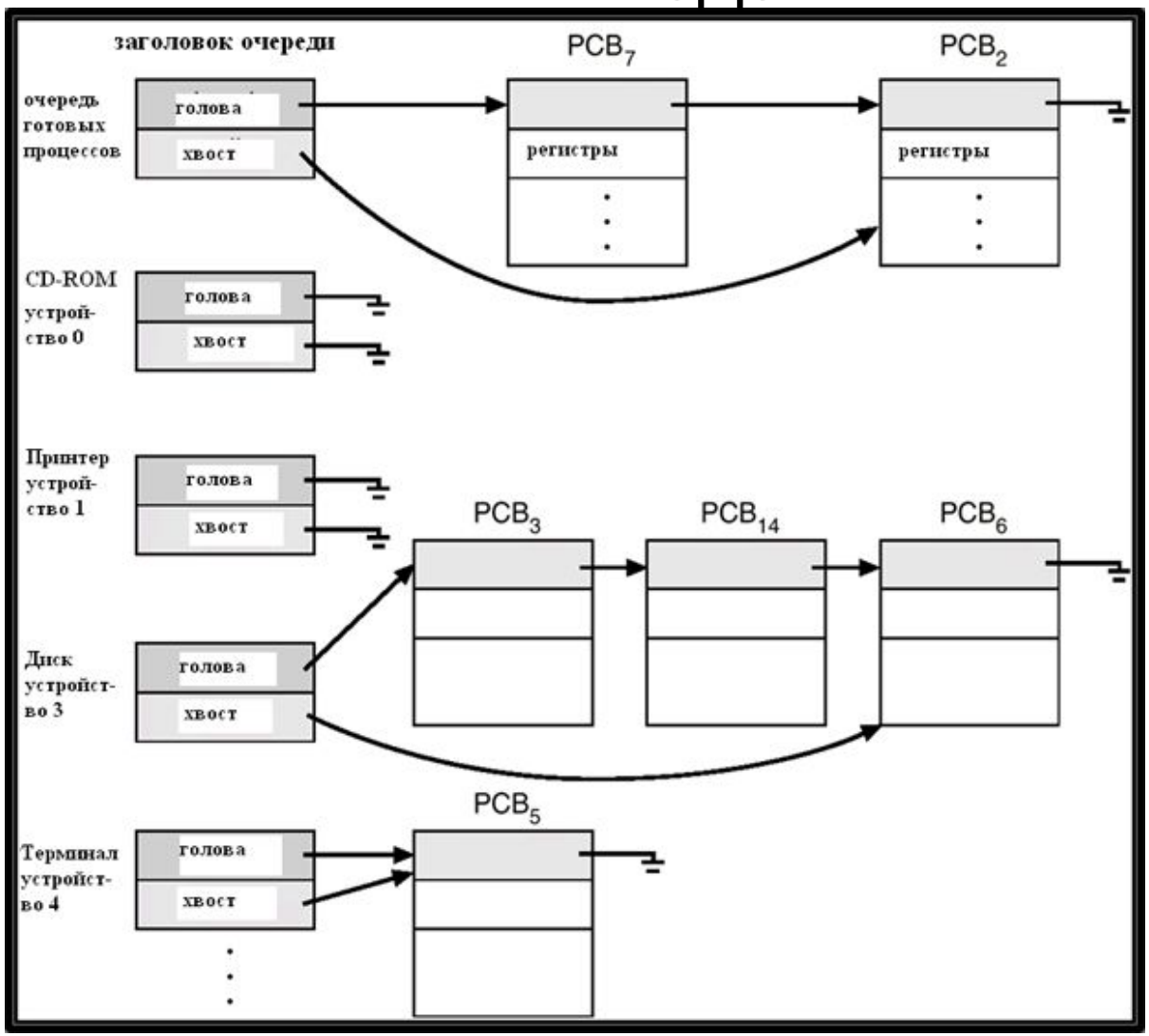
Блок управления процессом (PCB)



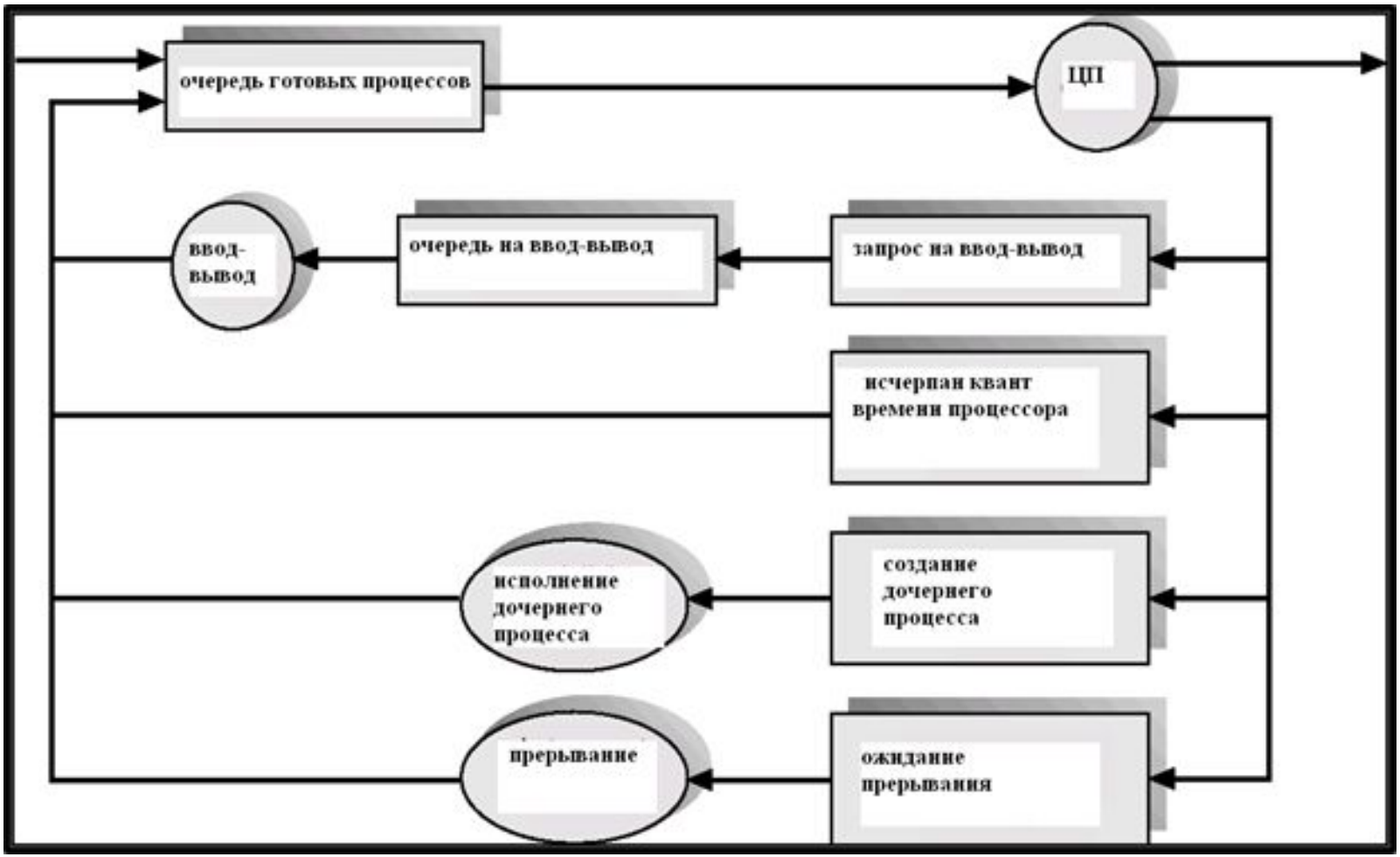
Переключение процессора с одного процесса на другой



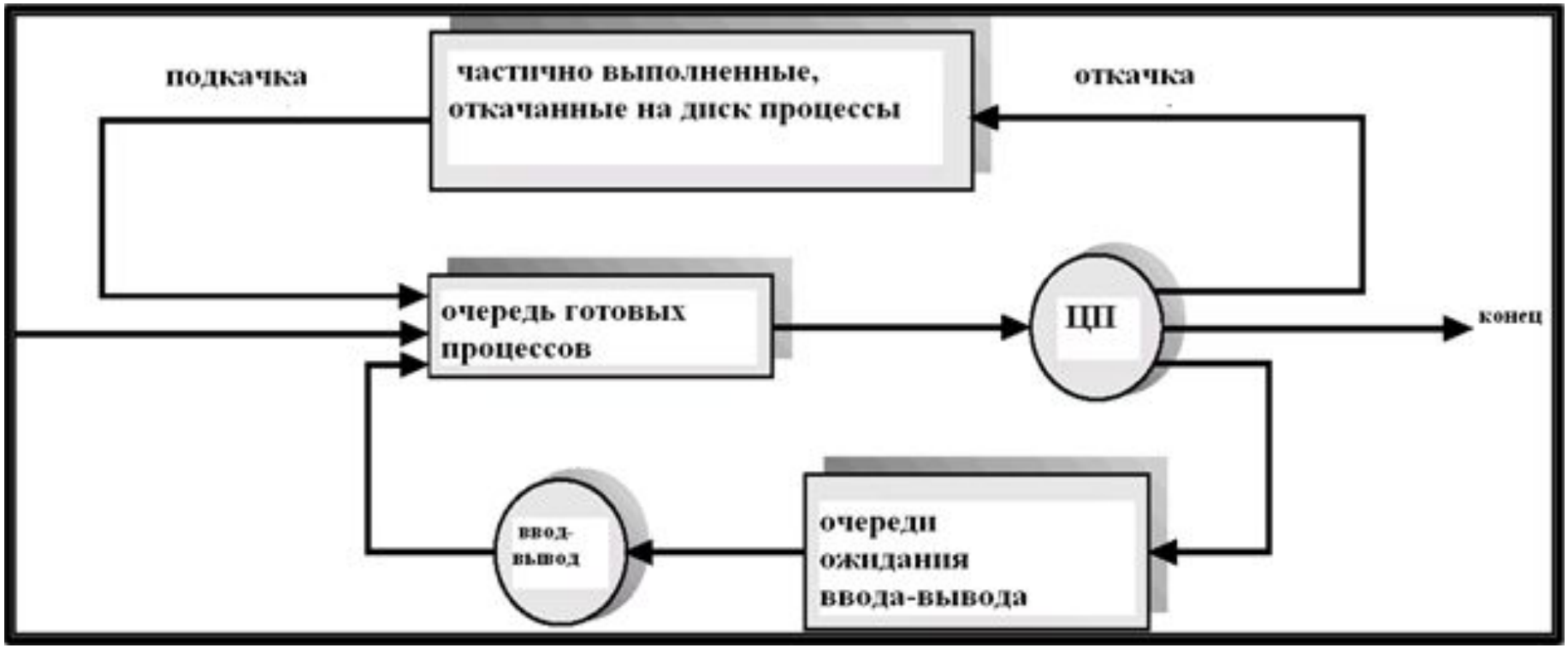
Очередь готовых процессов и очереди для различных устройств ввода-вывода



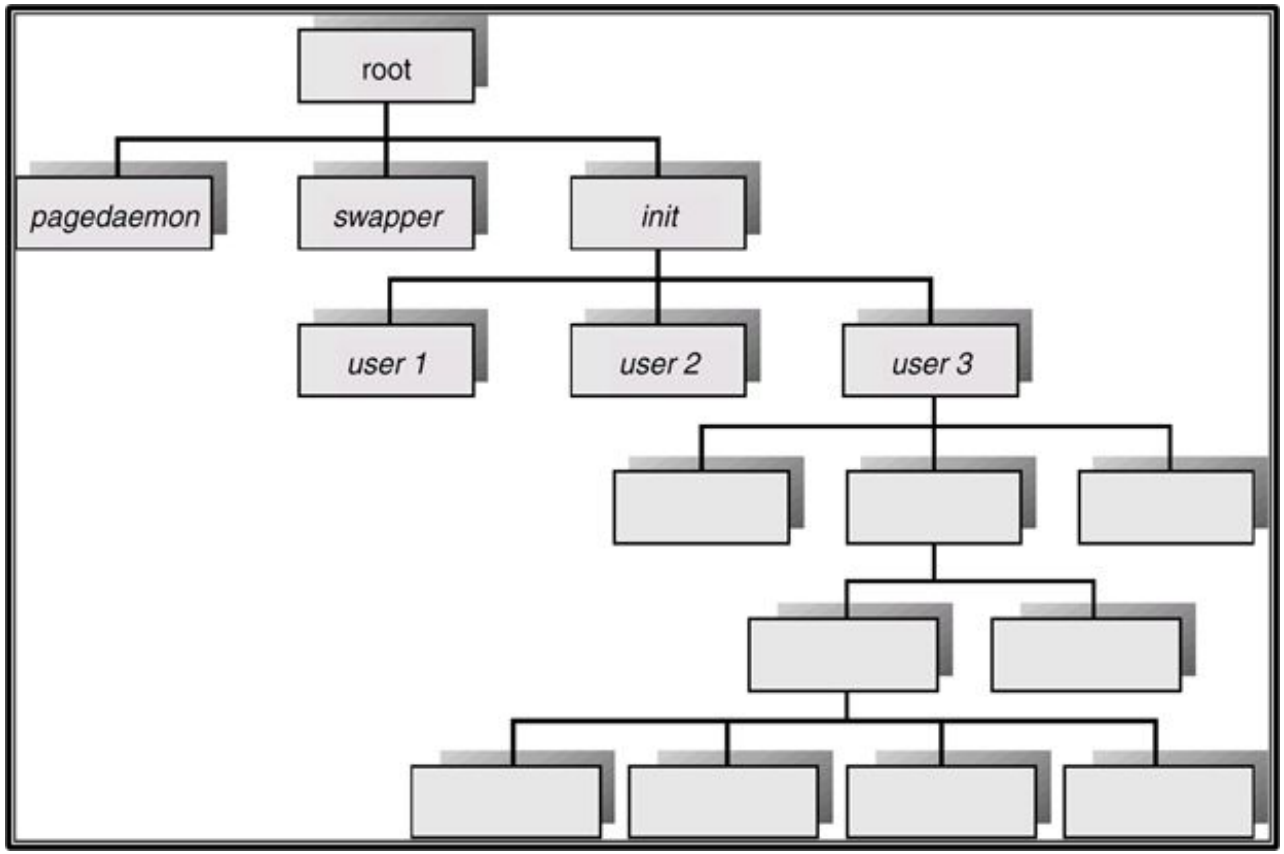
Графическое представление диспетчеризации процессов



Добавление диспетчера выгруженных процессов



Дерево процессов в системе UNIX



Ограниченный буфер – реализация с помощью общей памяти

- **Общие данные**

```
#define BUFFER_SIZE 1000 /* или другое
   конкретное значение */
 typedef struct {
. . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Ограниченный буфер: процесс-производитель

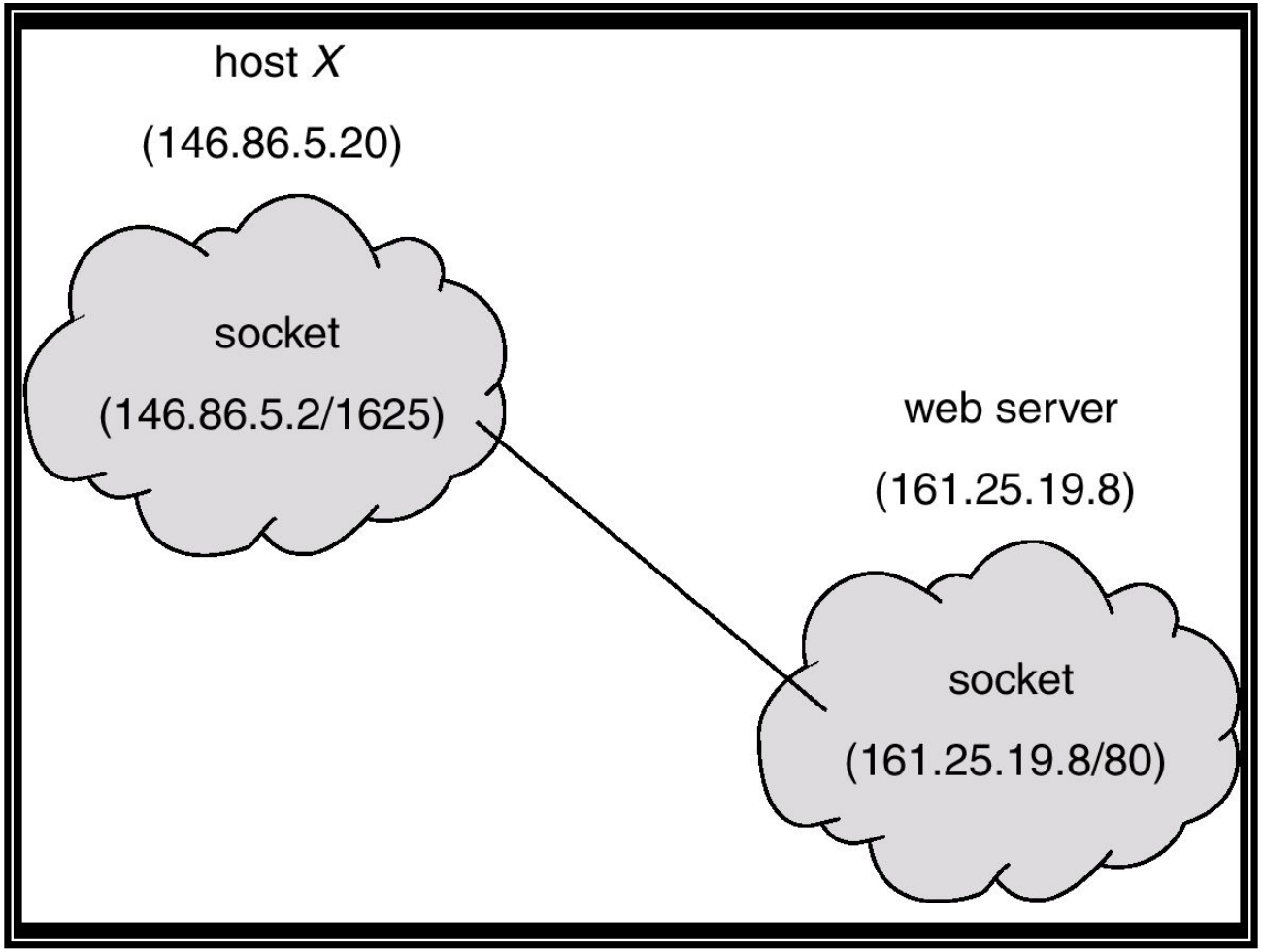
item nextProduced;

```
    item nextProduced; /* следующий генерируемый элемент */
while (1) { /* бесконечный цикл */
while (((in + 1) % BUFFER_SIZE) == out)
       ; /* ждать, пока буфер переполнен*/
buffer[in] = nextProduced; /* генерация элемента */
in = (in + 1) % BUFFER_SIZE;
}
```

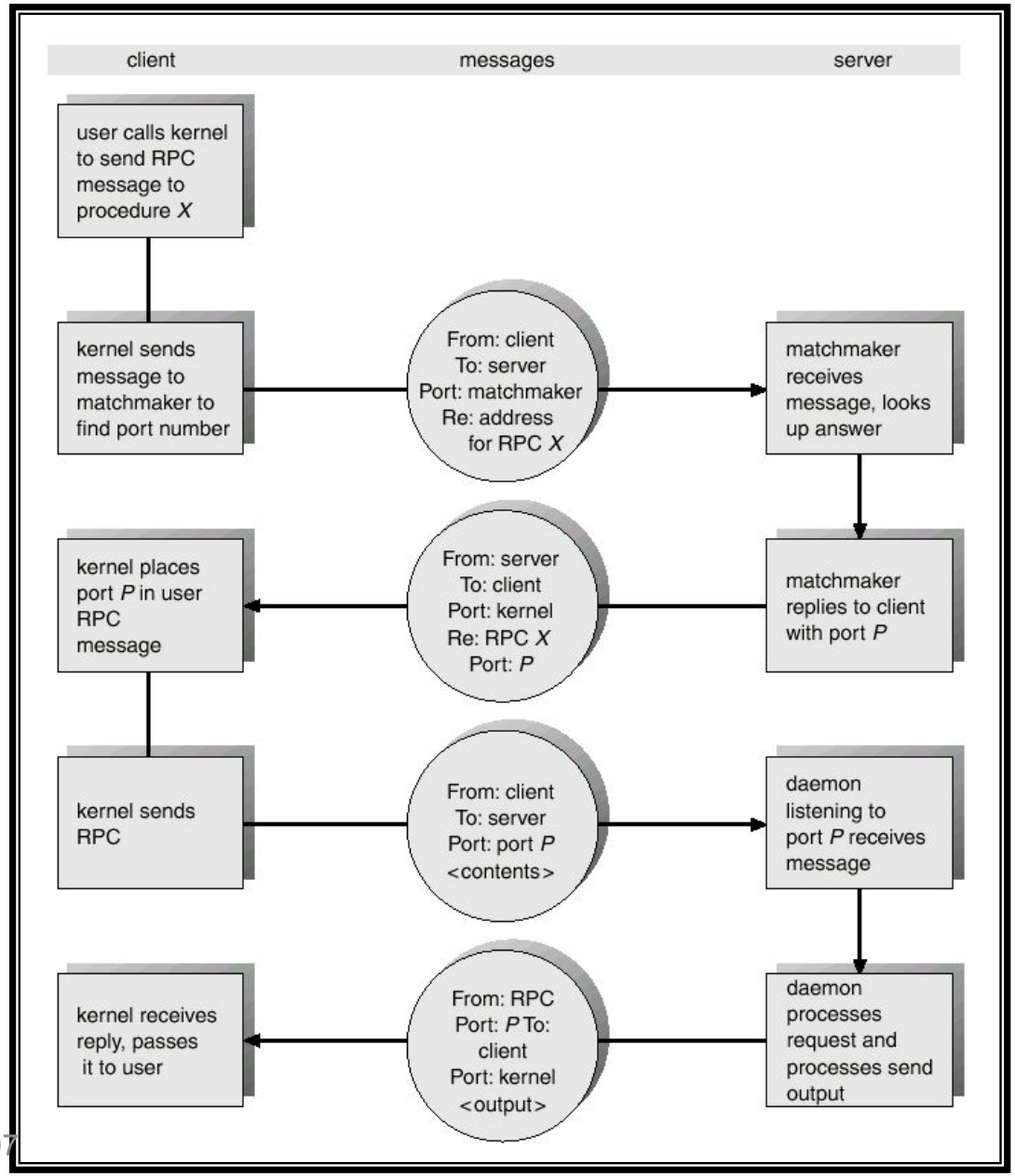
Ограниченный буфер: процесс-потребитель

```
    item nextConsumed; /* следующий используемый элемент */
while (1) { /* бесконечный цикл */
      while (in == out) ; /* ждать, пока буфер пуст */
      nextConsumed = buffer[out]; /* использование элемента */
      out = (out + 1) % BUFFER_SIZE;
}
```

Взаимодействие с помощью сокетов

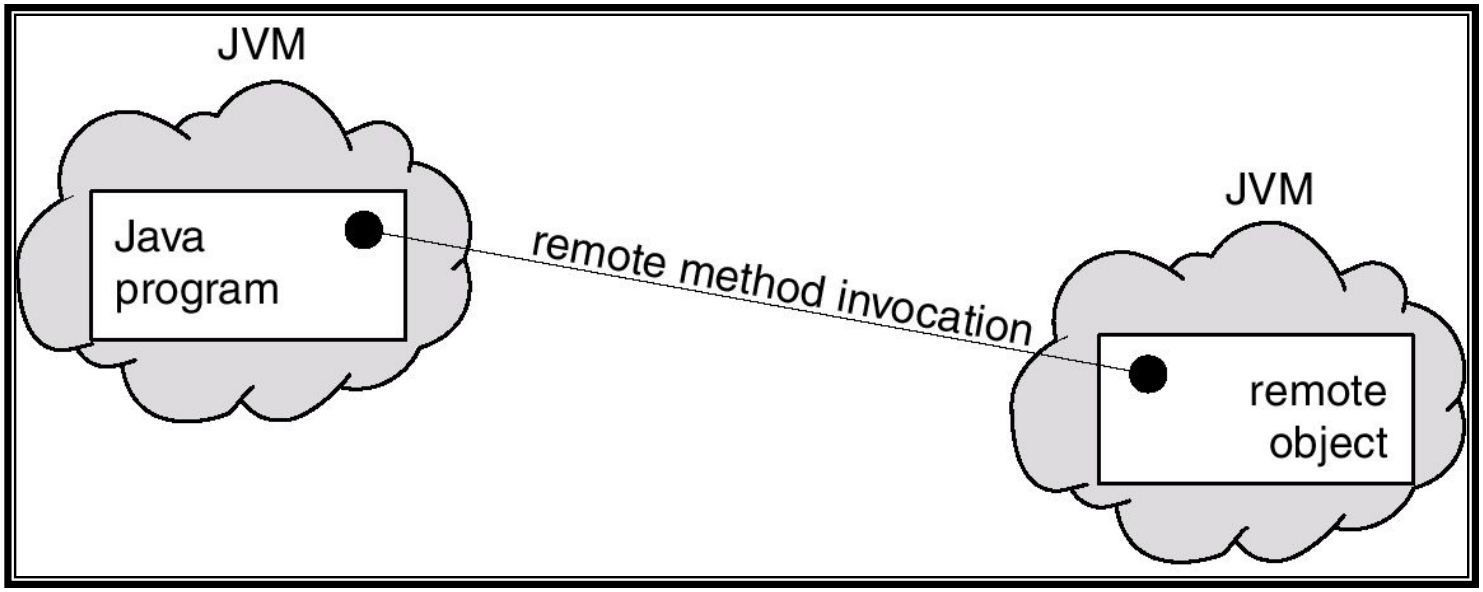


Исполнение RPC

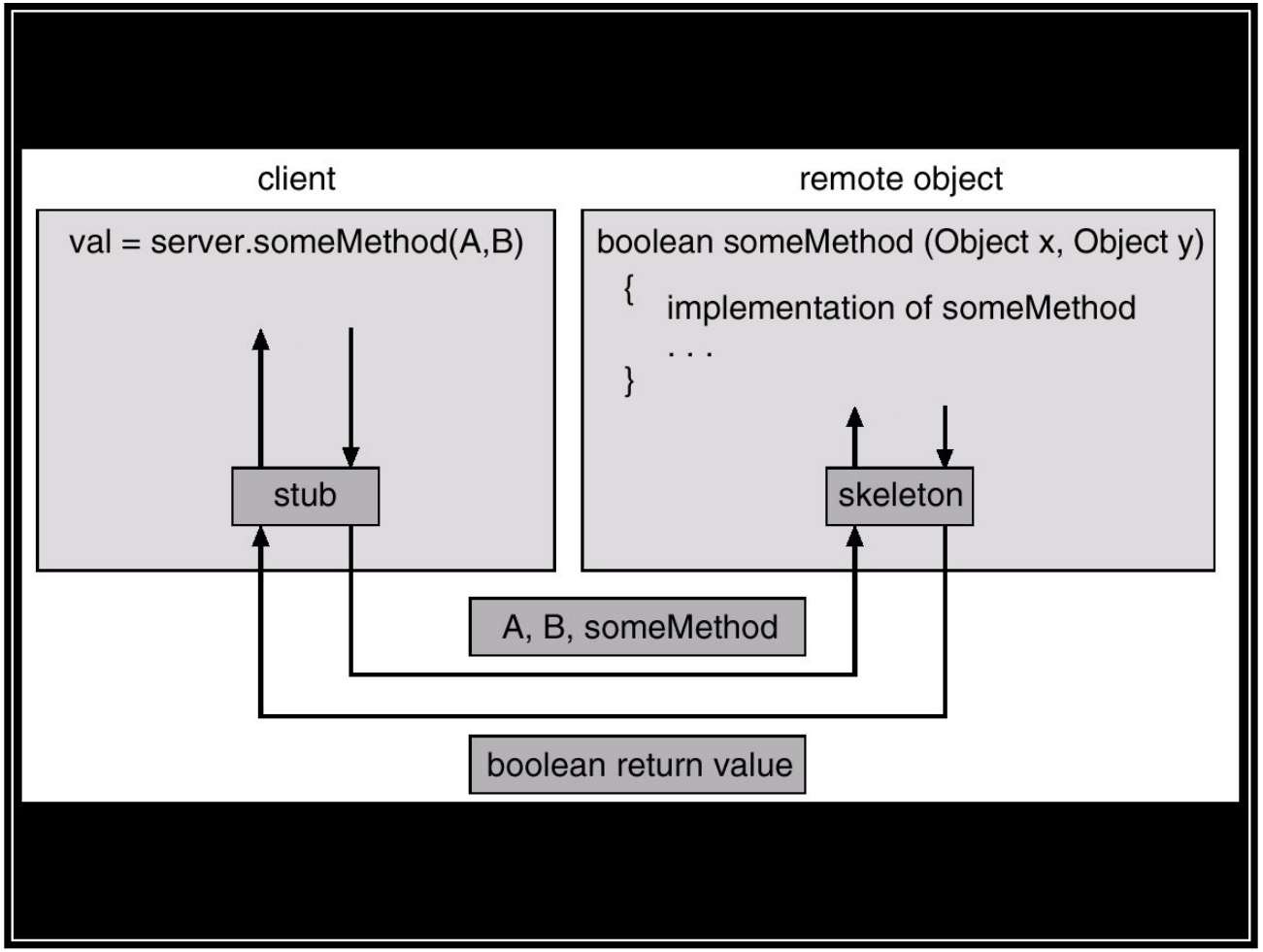


Удаленный вызов метода (RMI) - Java

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



Выстраивание параметров (marshaling)



Однопоточные и многопоточные процессы

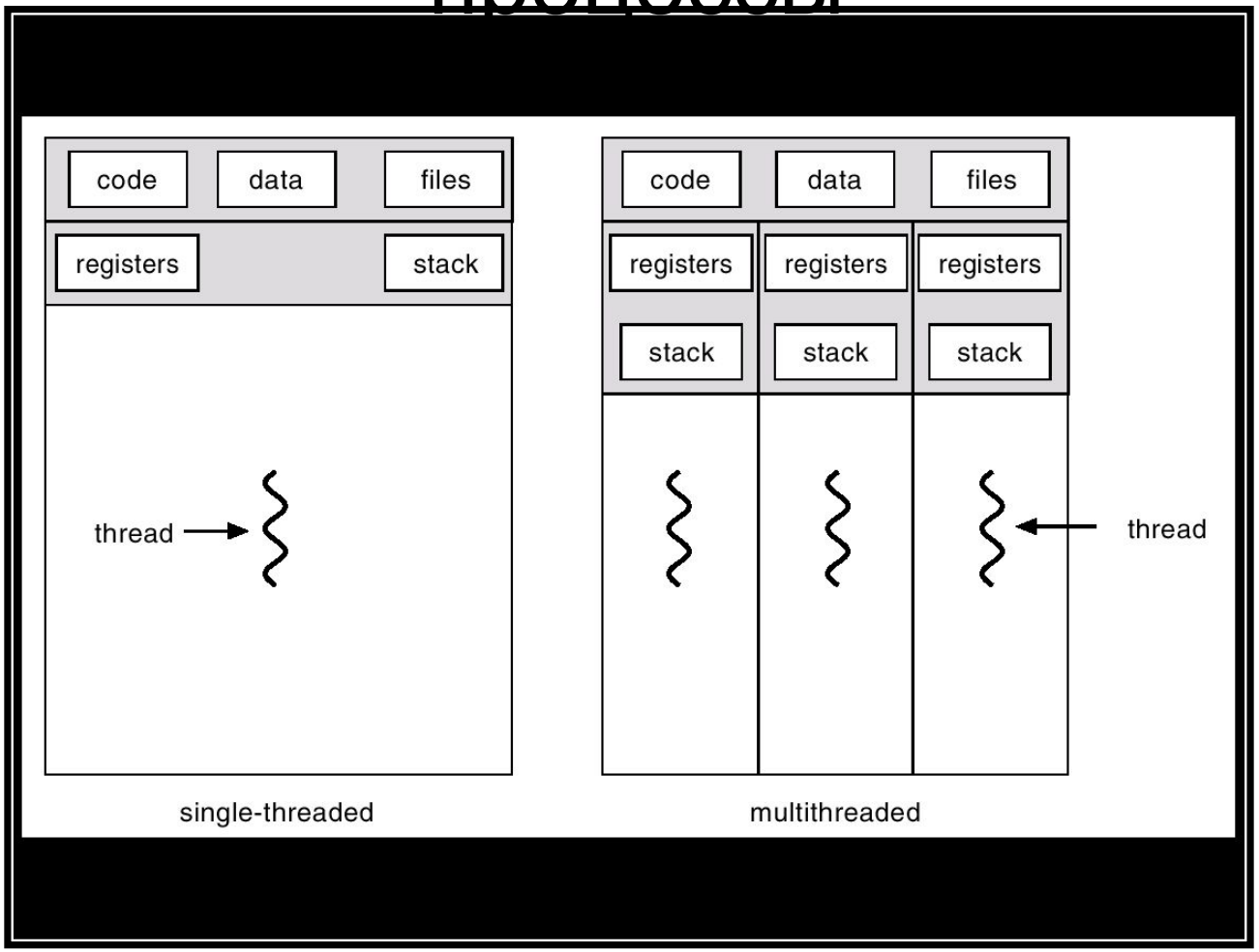


Схема модели “много / один”

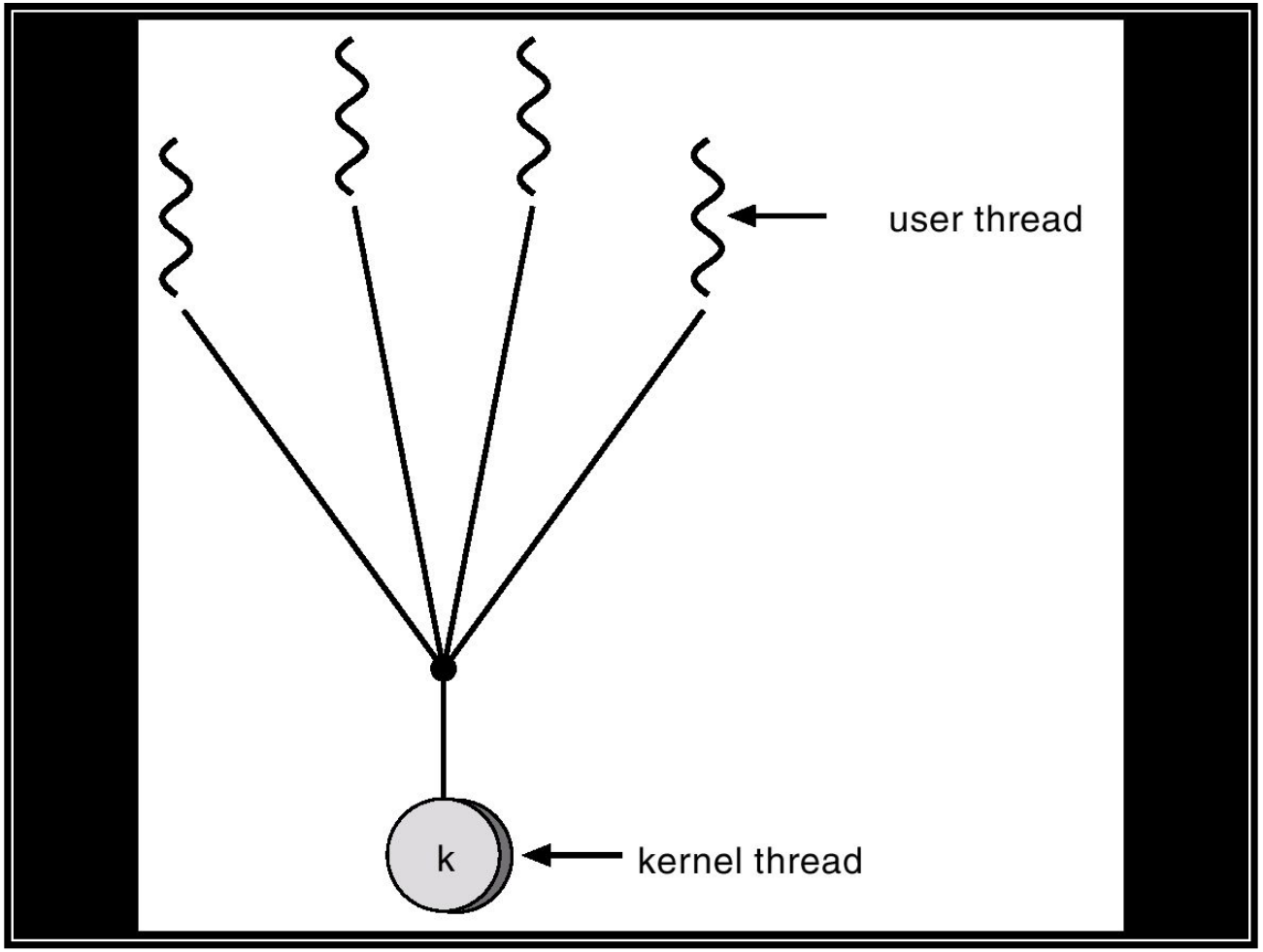


Схема модели “один / один”

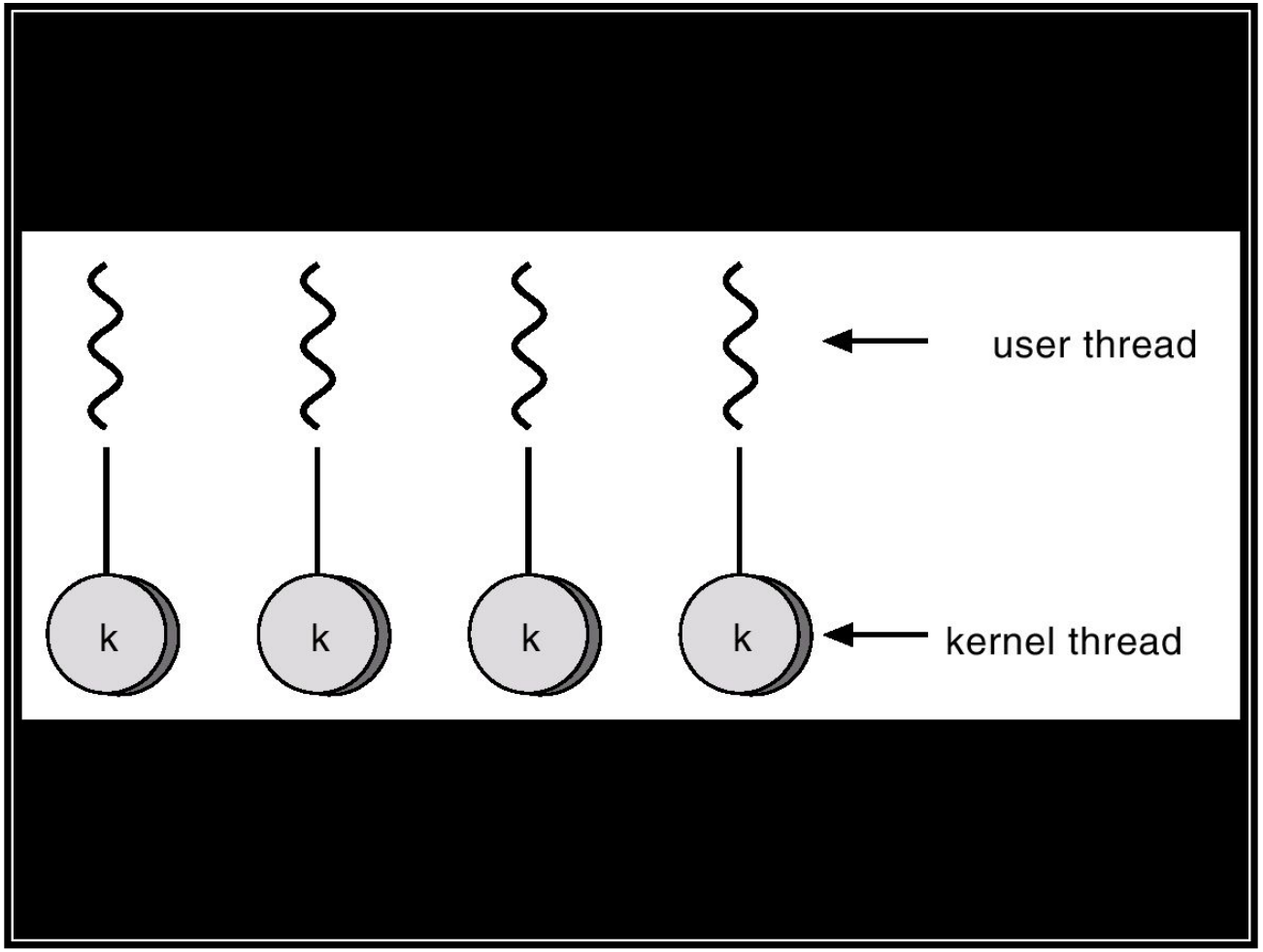
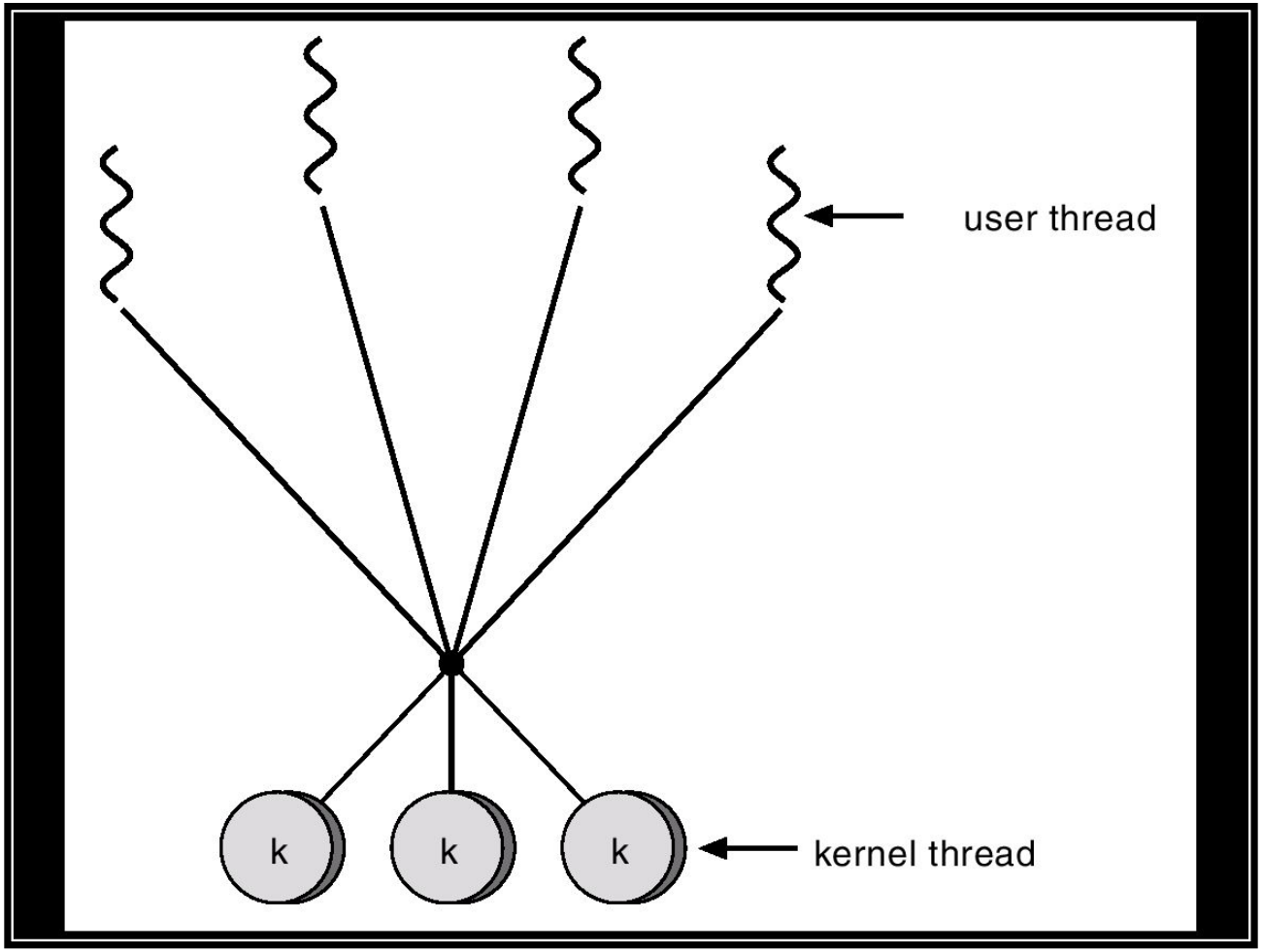
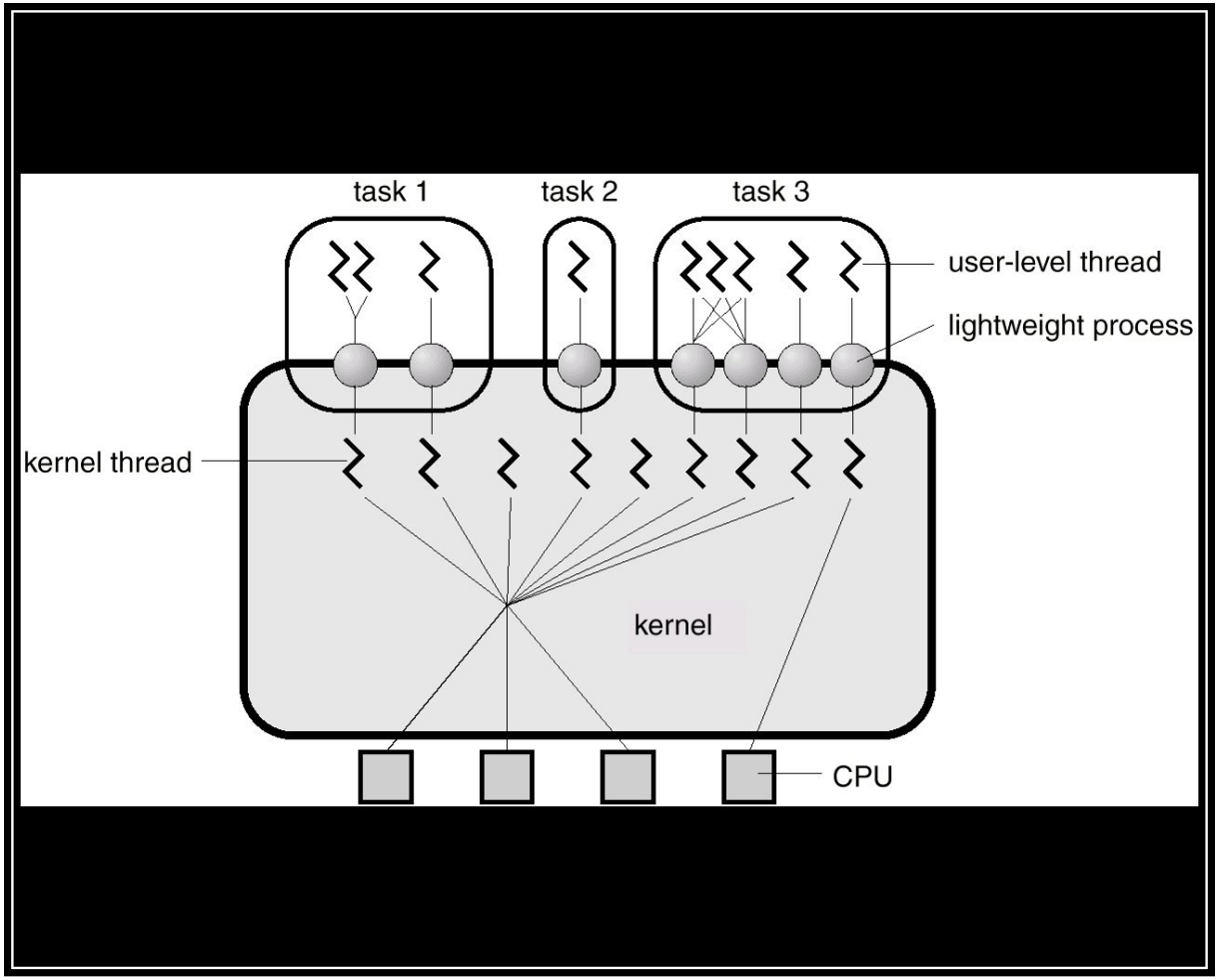


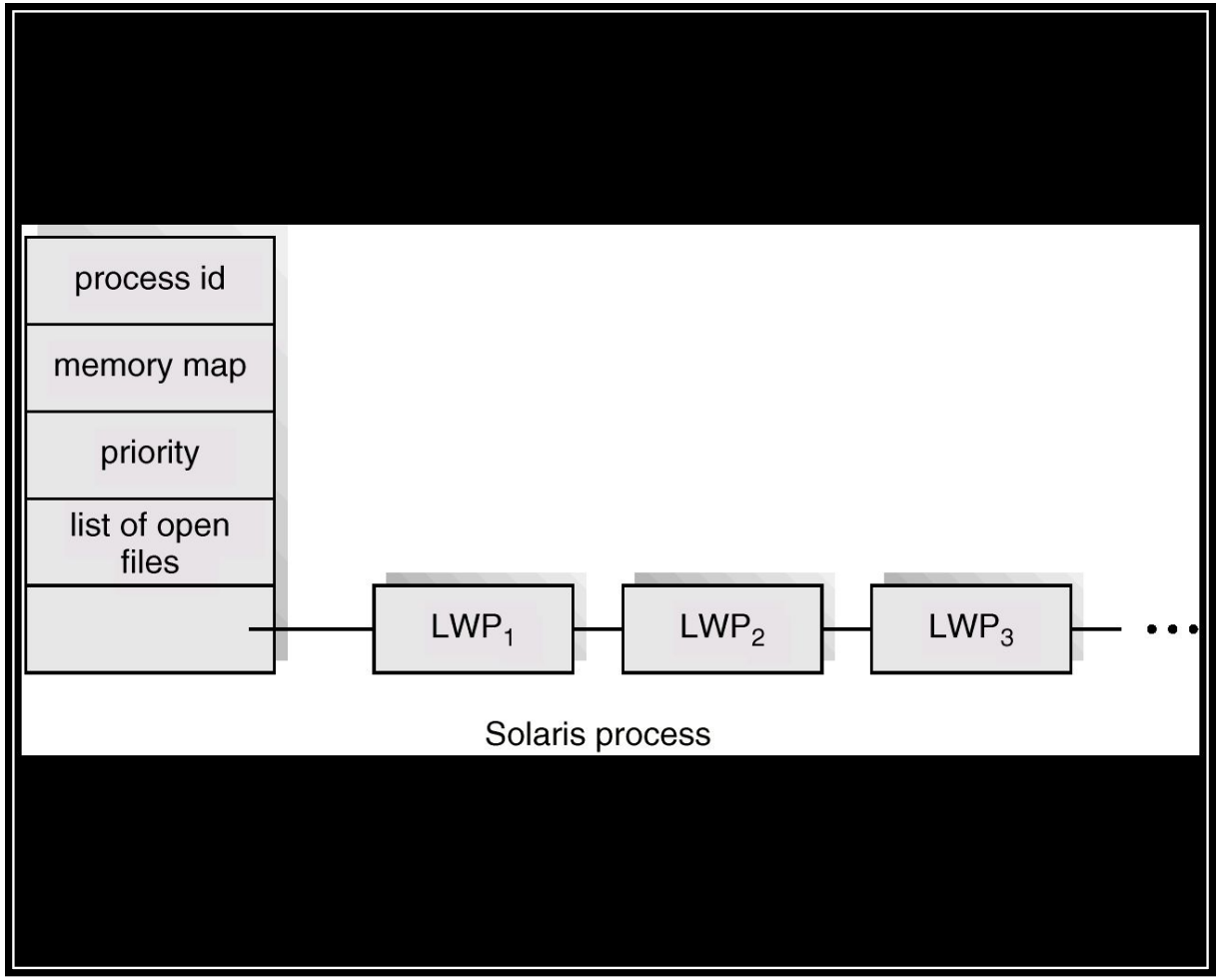
Схема модели “много/много”



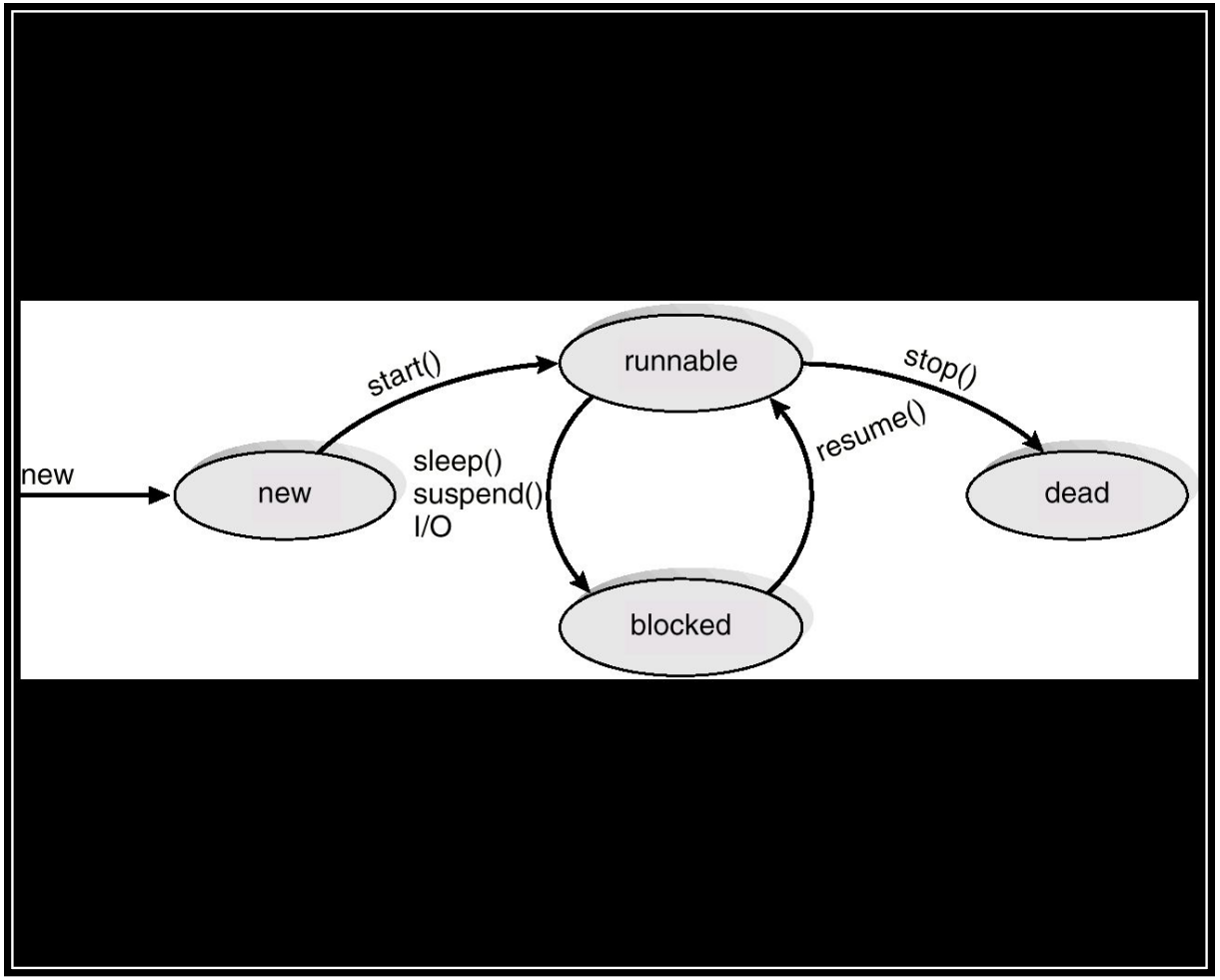
Потоки в Solaris



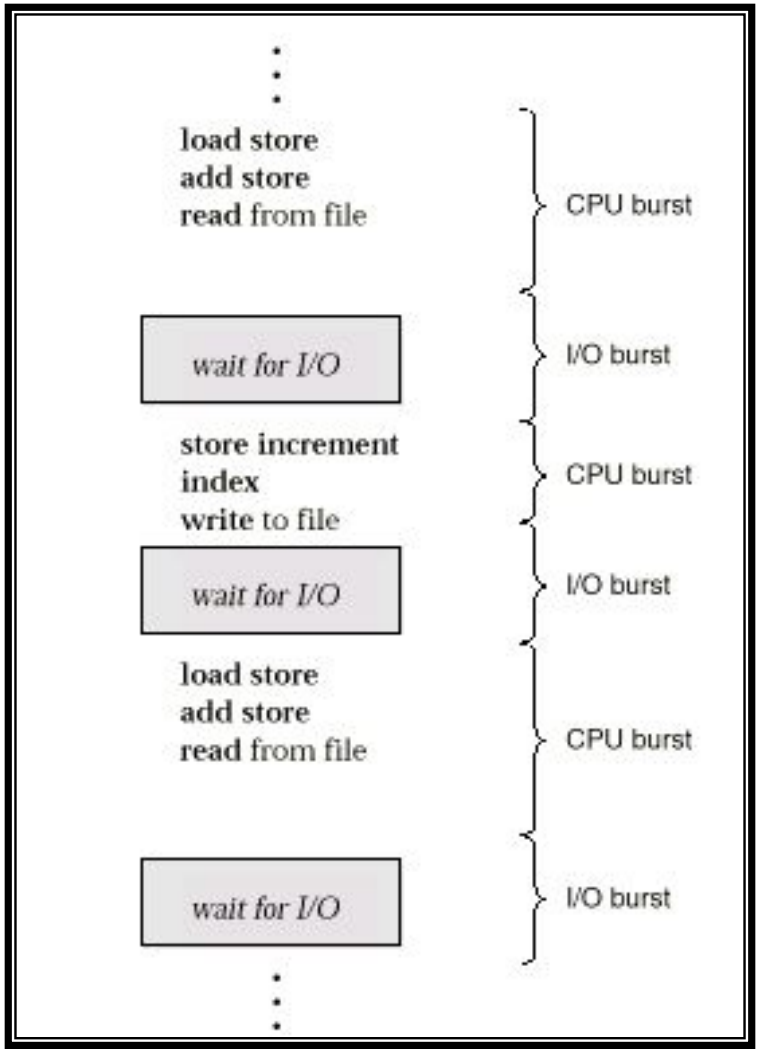
Процесс в Solaris



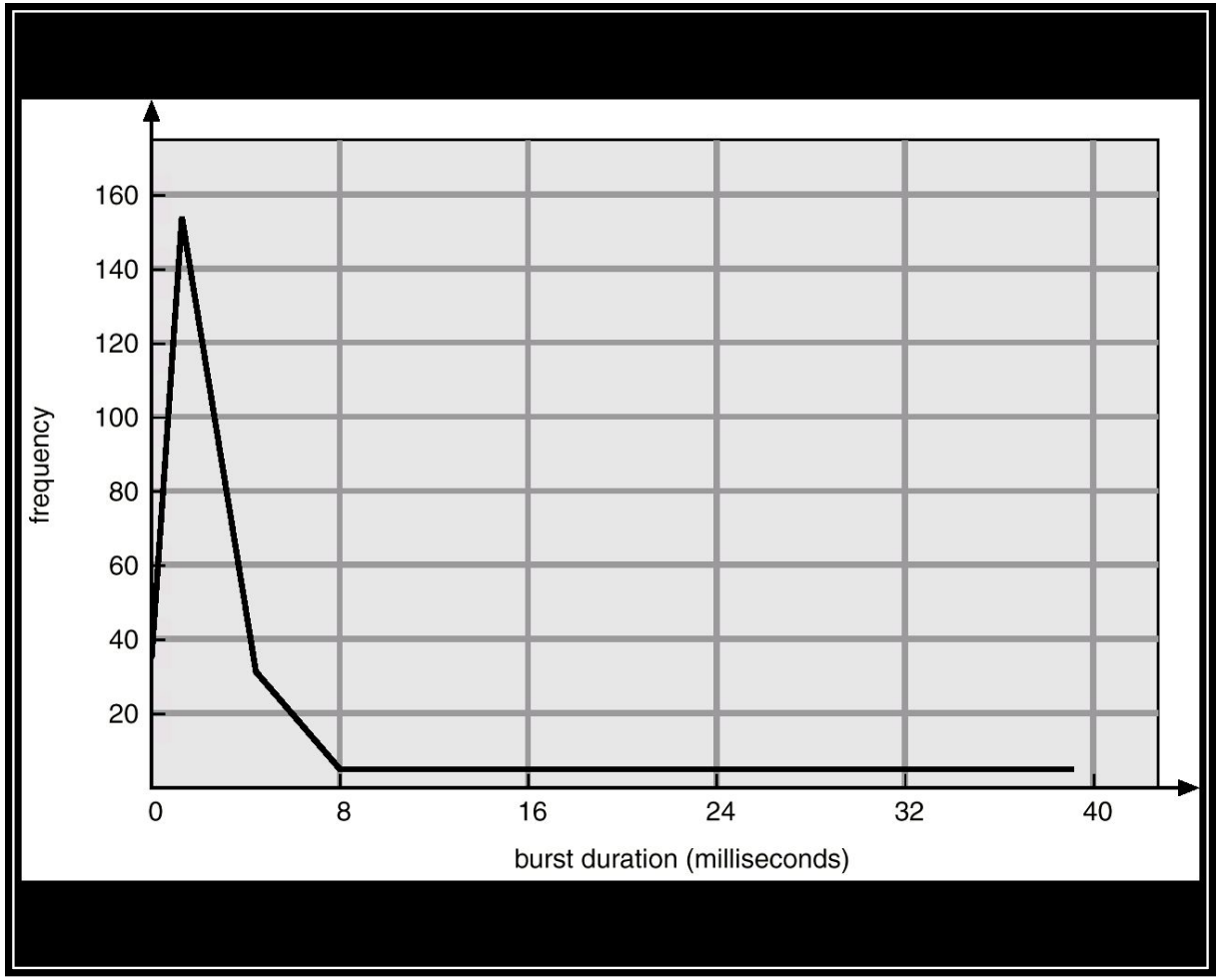
Состояния потоков в Java



Последовательность активных фаз (bursts) процессора и ввода-вывода



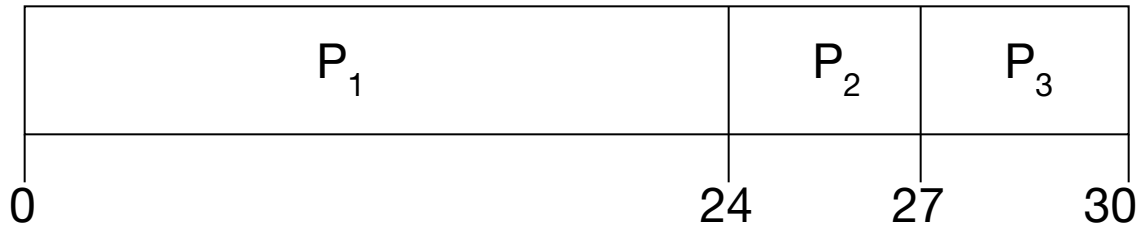
Гистограмма периодов активности процессора



Стратегия диспетчеризации
First-Come-First-Served (FCFS)

Процесс	Период активности
P_1	24
P_2	3
P_3	3

- **Пусть порядок процессов таков: P_1 , P_2 , P_3**
  Диаграмма Ганта (Gantt Chart) для их распределения:



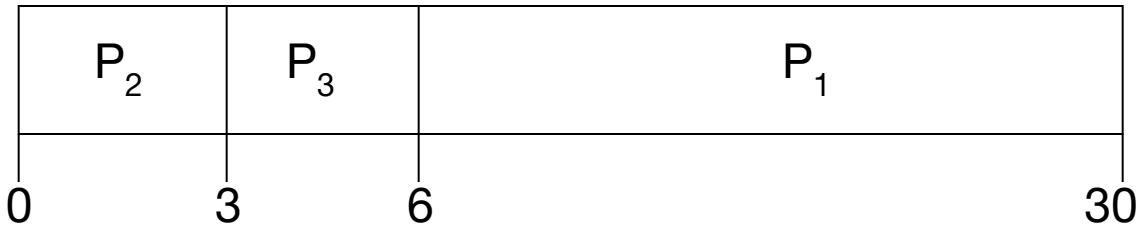
- Время ожидания для P_1 = 0; P_2 = 24; P_3 = 27
- Среднее время ожидания: (0 + 24 + 27)/3 = 17

Стратегия FCFS (продолжение)

Пусть порядок процессов таков:

P_2, P_3, P_1.

- **Диаграмма Ганта для их распределения:**

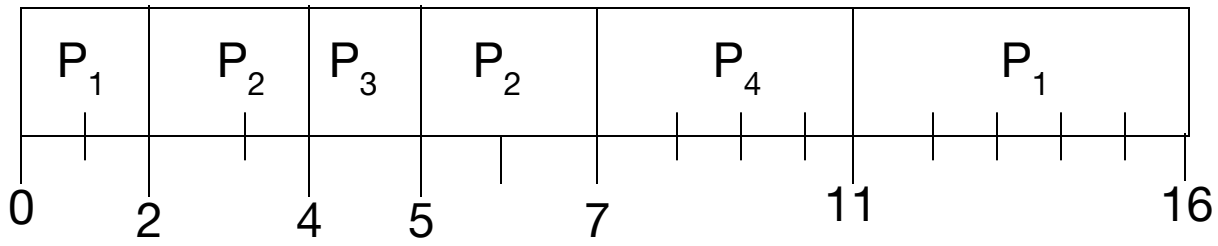


- Время ожидания: $P_1 = 6; P_2 = 0; P_3 = 3$
- Среднее время ожидания: $(6 + 0 + 3)/3 = 3$
- Много лучше, чем в предыдущем случае.
- *Эффект сопровождения (convoy effect)* - короткий процесс после долгого процесса

Пример: SJF с опережением

Процесс	Время появления	Время активности
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- **SJF (с опережением)**



- **Среднее время ожидания = (9 + 1 + 0 +2)/4 = 3**

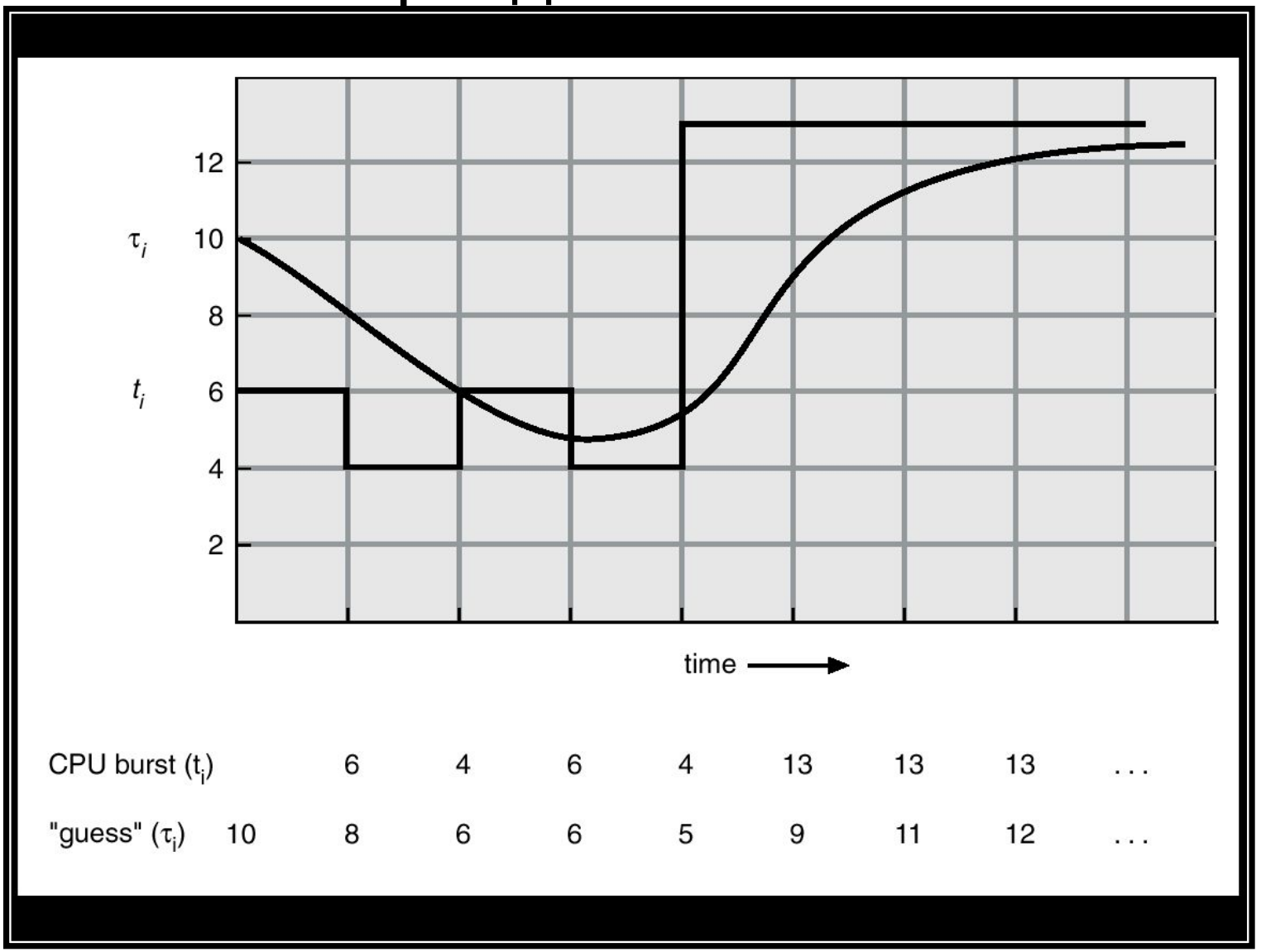
Определение длины следующего периода активности

- **Является лишь оценкой длины.**
- **Может быть выполнено с использованием длин предыдущих периодов активности, используя экспоненциальное усреднение**

1. $t_n = $ actual lenght of n^{th} CPU burst
2. $\tau_{n+1} = $ predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define :

$$\tau_{n=1} = \alpha\ t_n + (1-\alpha)\tau_n .$$

Предсказание длины следующего периода активности



Примеры экспоненциального усреднения

- **α =0**
  - **$\tau_{n+1} = \tau_n$**
  - **Недавняя история не учитывается.**
- **α =1**
  - **$\tau_{n+1} = t_n$**
  - **Учитывается только фактическая длина последнего периода активности.**
- **Если обобщить формулу, получим:**

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\,\alpha\, t_{n-1} + \ldots + (1 - \alpha)^j \alpha\, t_{n-1} + \ldots + (1 - \alpha)^{n+1} t_n\, \tau_0$$

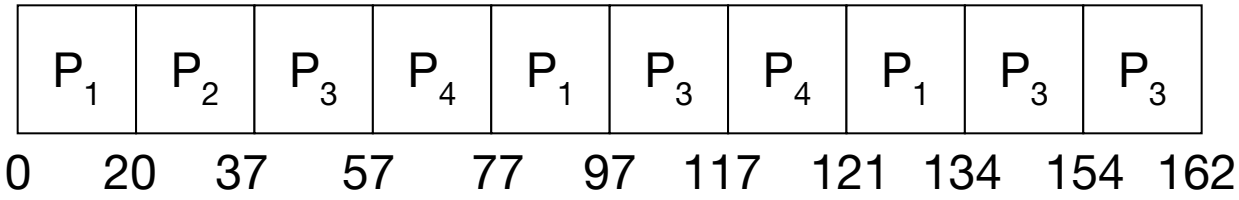
- **Так как α и(1 - α) не превосходят 1, каждый последующий терм имеет меньший вес, чем его предшественник**

Пример RR (квант времени = 20)

- **Пример RR с квантом времени = 20**

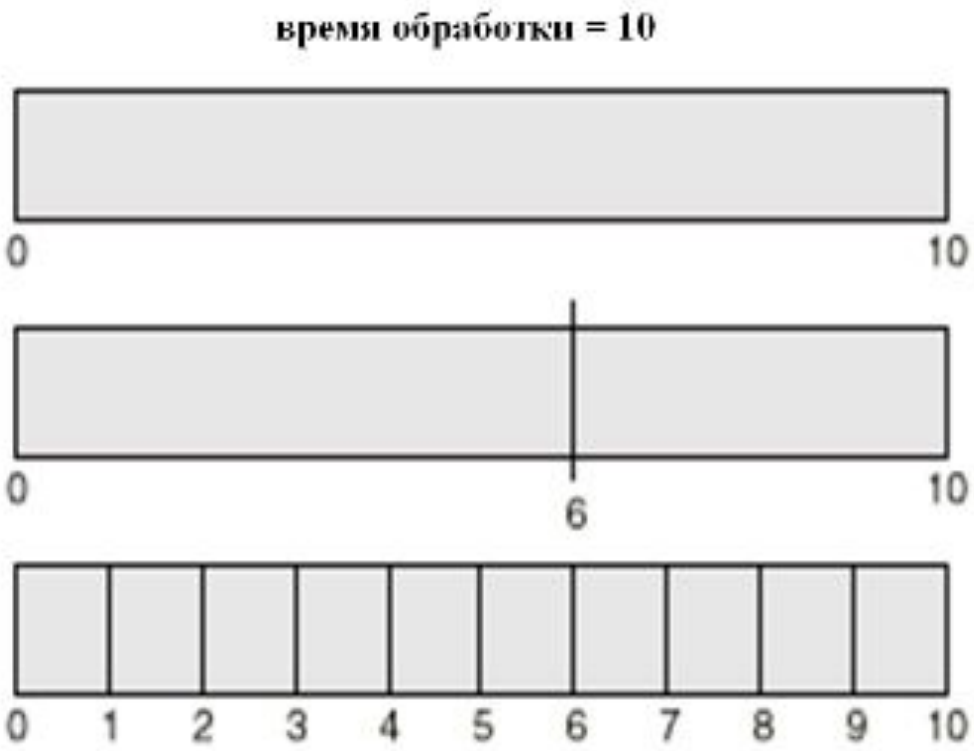
Процес	**Время активности**
P_1	53
P_2	17
P_3	68
P_4	24

- **Диаграмма Ганта:**



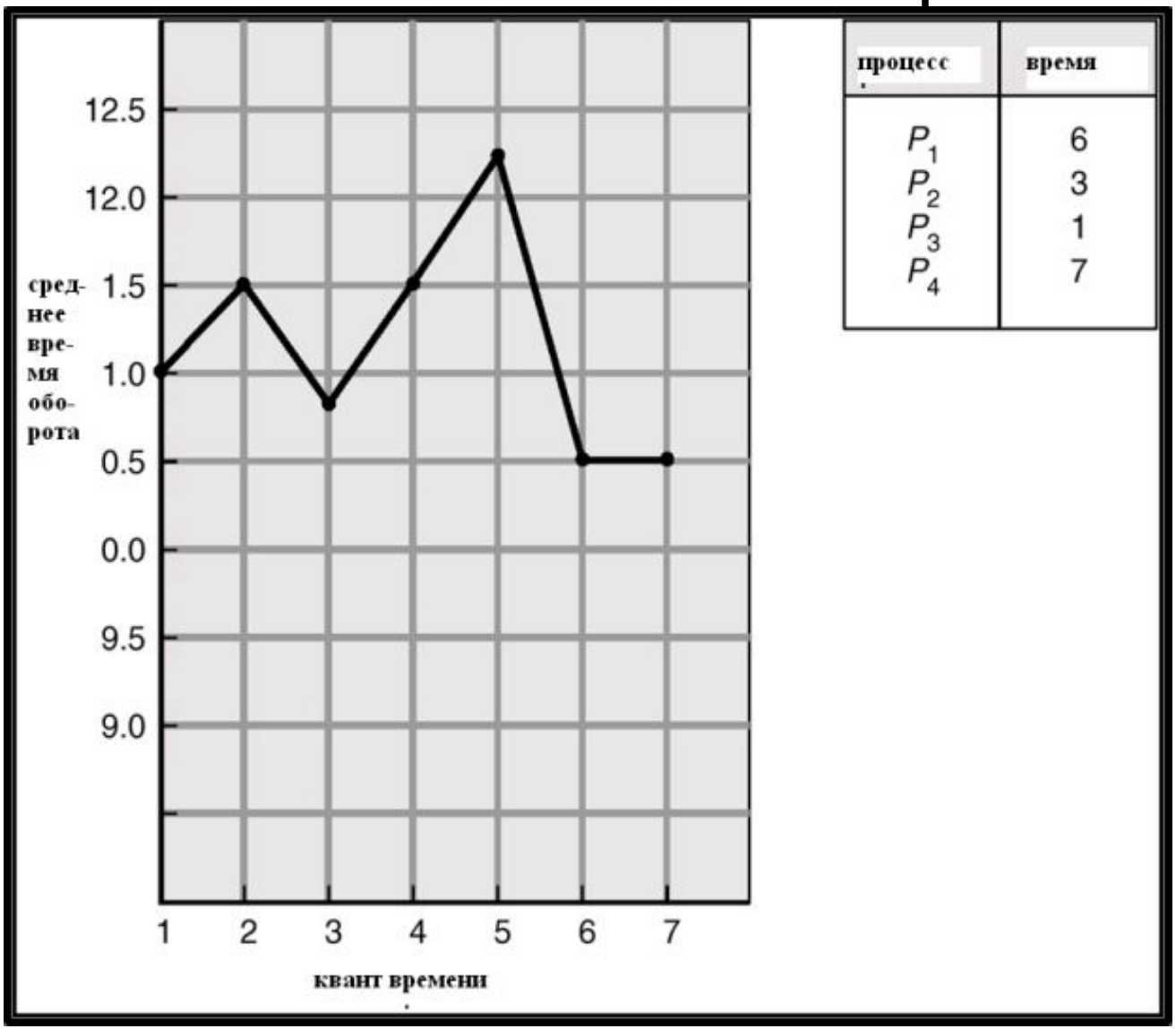
- **Обычно RR имеет худшее время оборота, чем SJF, но лучшее время ответа.**

Квант времени ЦП и время переключения контекста



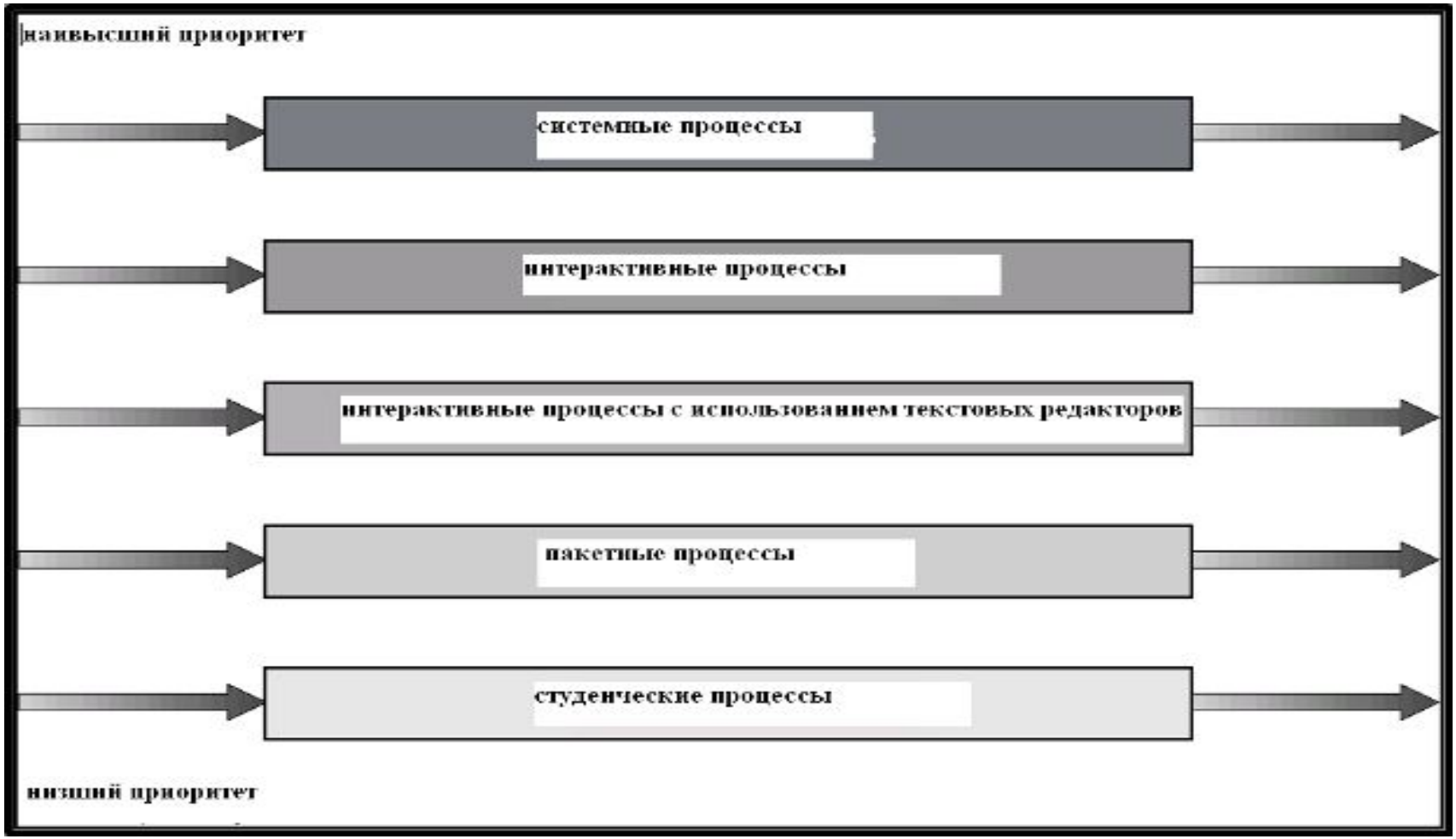
квант	контекстные переключения
12	
6	1
1	9

Изменение времени оборота, в зависимости от кванта времени

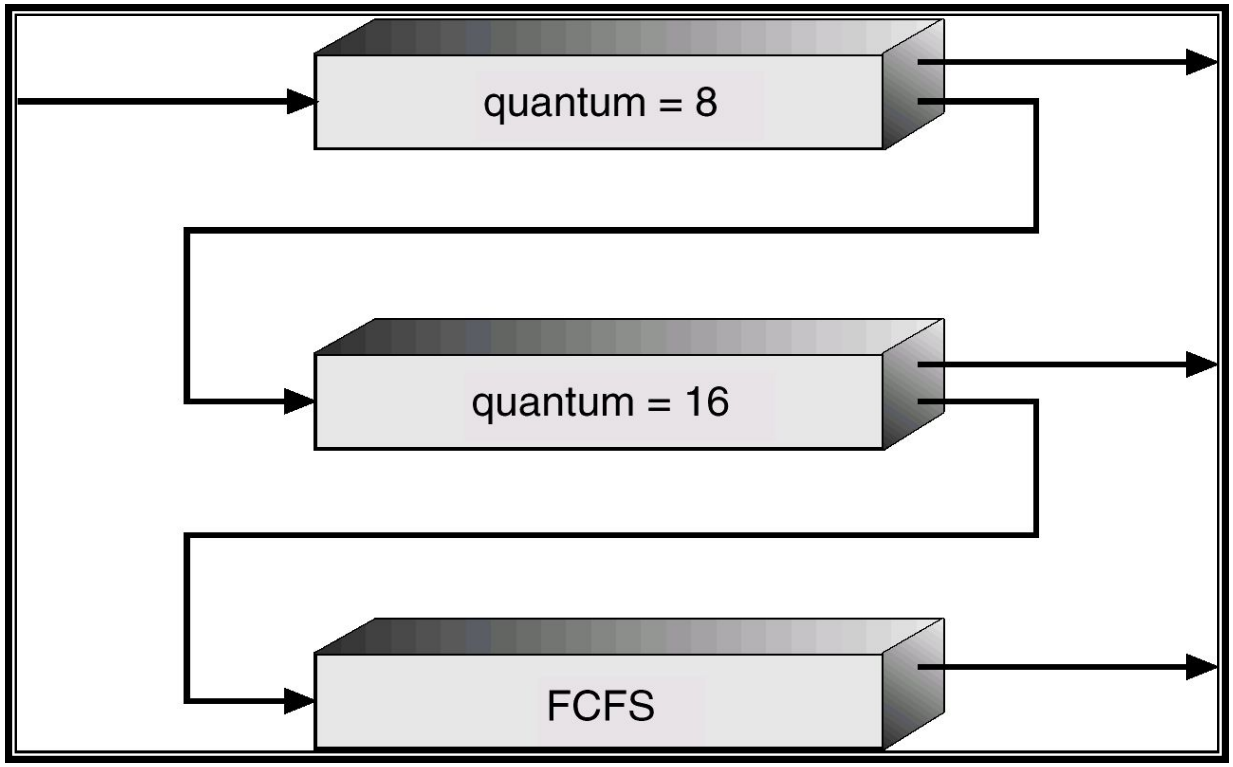


процесс	время
P_1	6
P_2	3
P_3	1
P_4	7

Диспетчеризация по принципу многоуровневой очереди



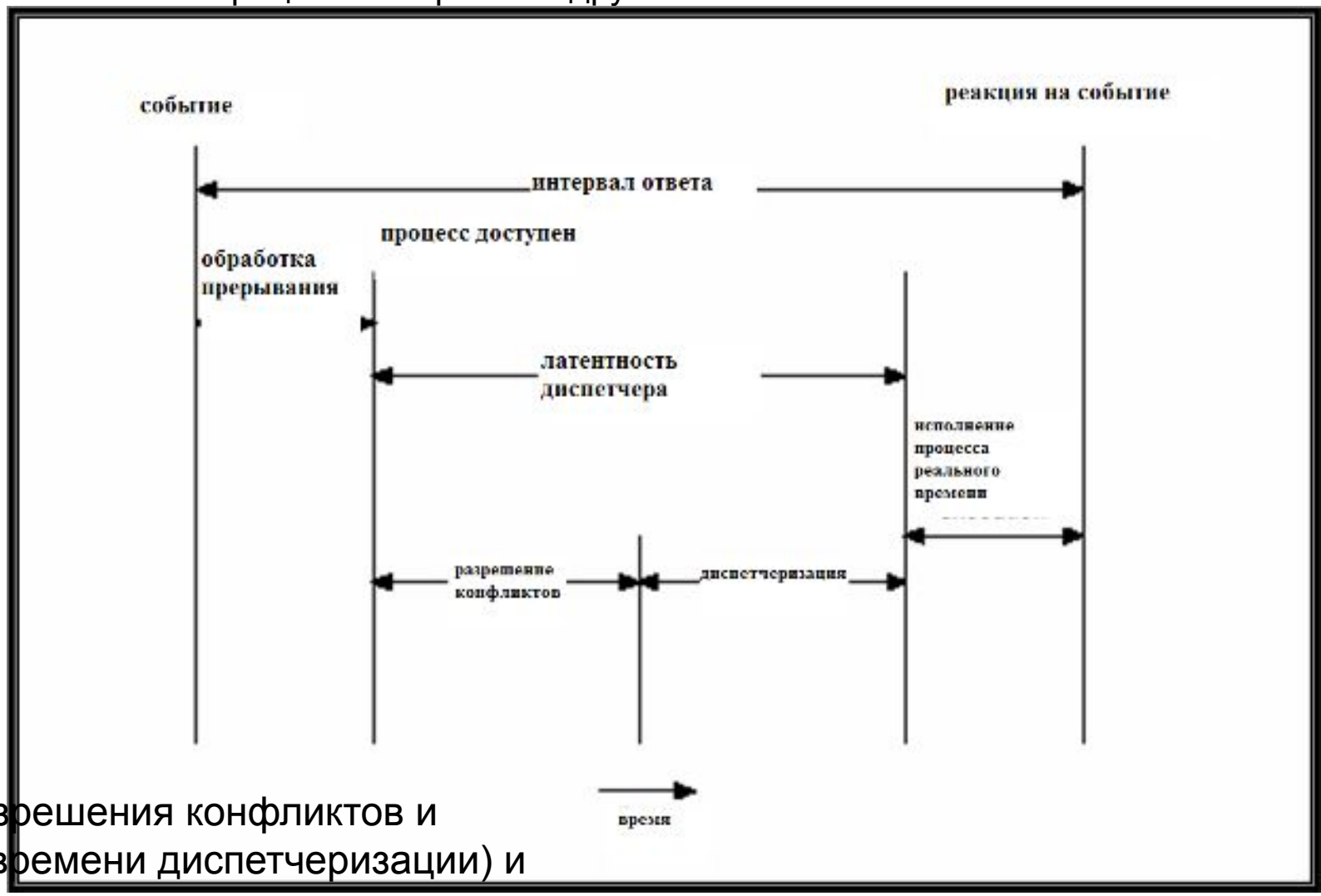
Многоуровневые аналитические очереди



кванты времени 8 (очередь Q0) и 16 (очередь Q1) и пакетными процессами по стратегии FCFS (очередь Q2). Первоначально процесс помещается в очередь Q0; если он не завершается за 8 единиц времени, то он перемещается в очередь Q1; если не завершается и за 16 единиц времени – то перемещается в очередь Q2.

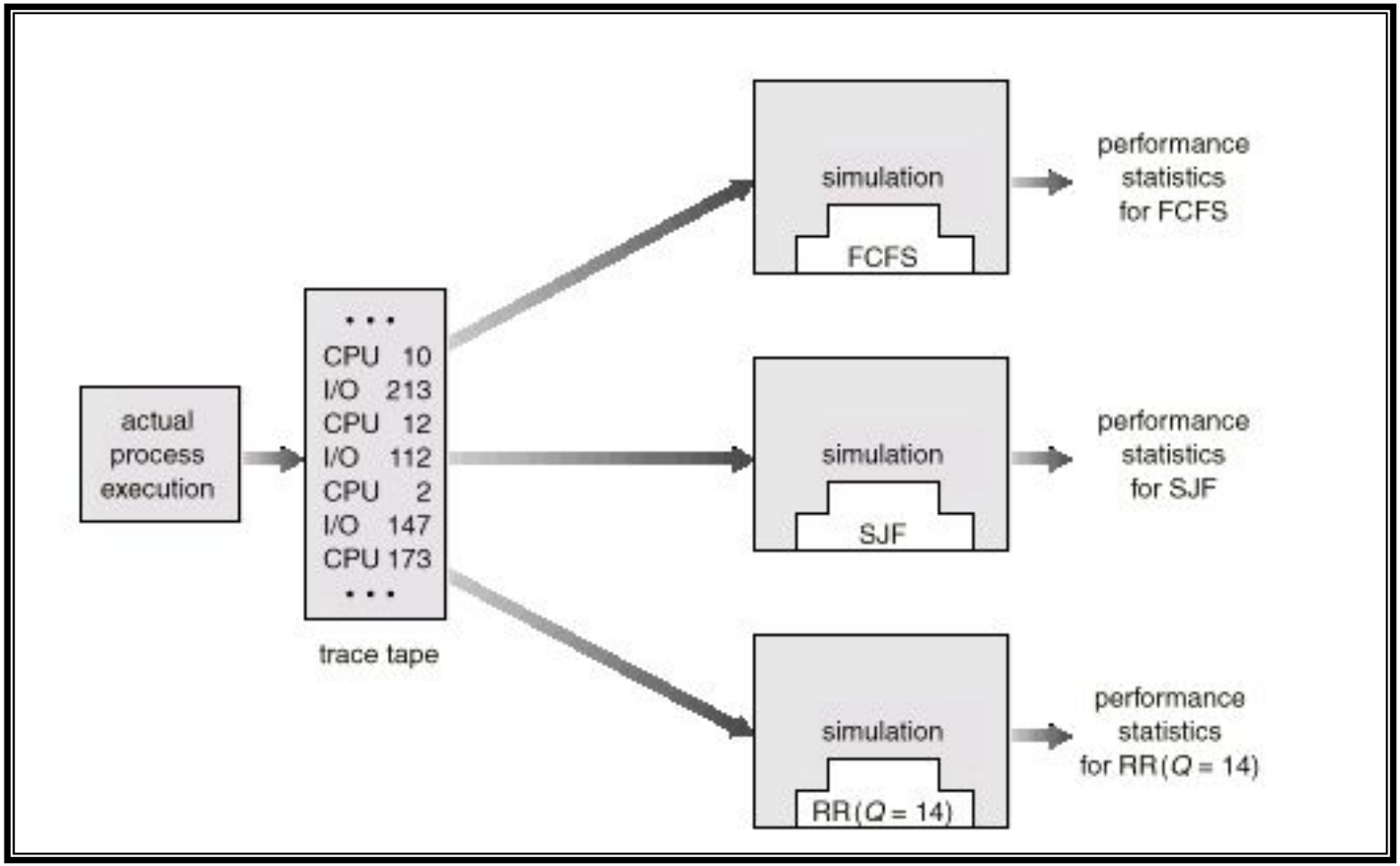
Латентность диспетчера
(dispatch latency)– время, требуемое для диспетчера, чтобы остановить один процесс и стартовать другой.

Интервал ответа, который не может быть превышен, складывается из времени обработки прерывания, периода латентности диспетчера при переключ контекста

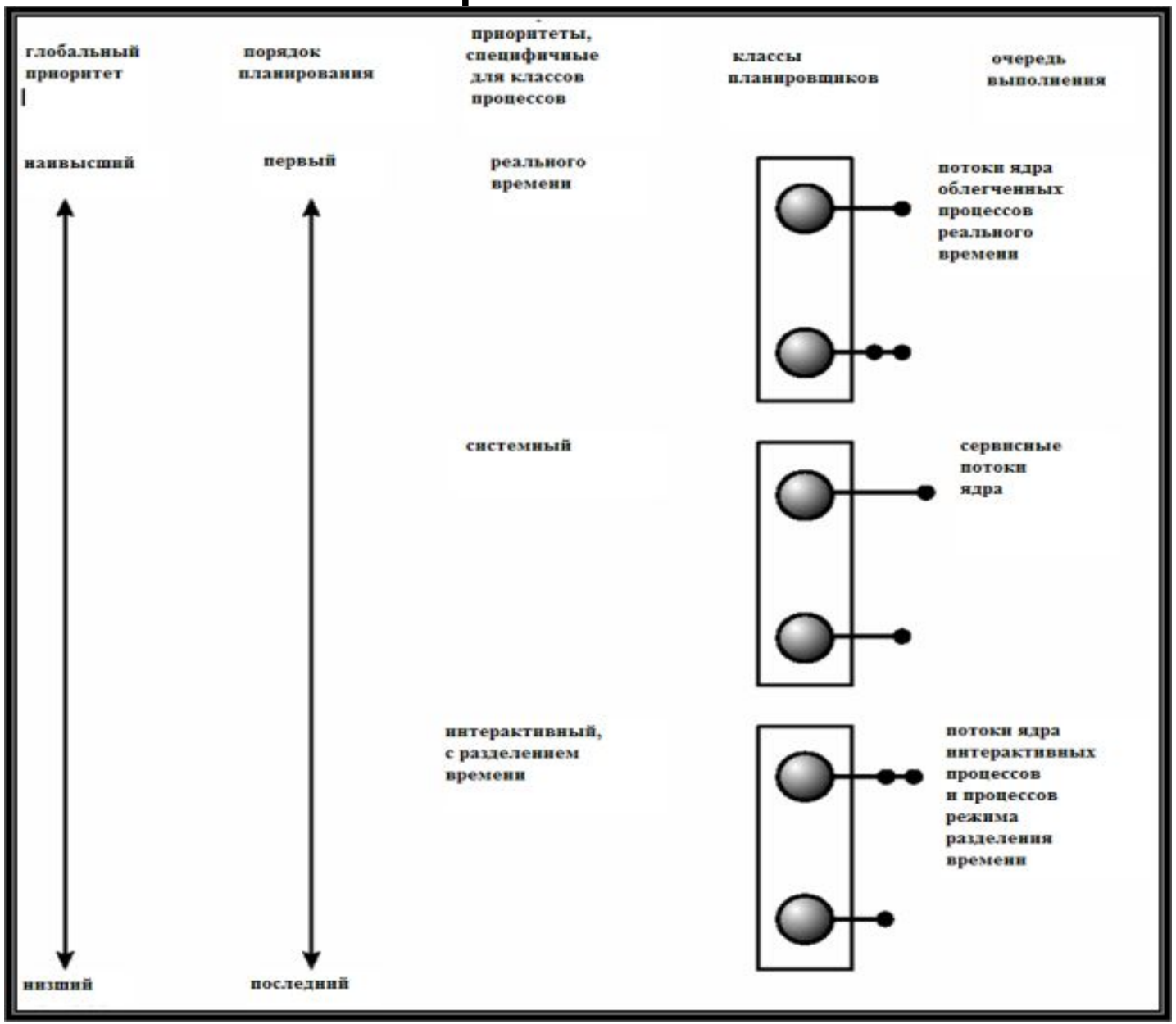


(времени разрешения конфликтов и собственно времени диспетчеризации) и времени исп критич проц реального времени.

Оценка планировщиков ЦП посредством моделирования



Планирование в Solaris



Приоритеты в Windows NT 5 и выше

Таблица 1.

	реального времени	высокий	выше нормального	нормальный	ниже нормального	приоритет простаивающего процесса
критический	31	15	15	15	15	15
наивысший	26	15	12	10	8	6
выше нормального	25	14	11	9	7	5
нормальный	24	13	10	8	6	4
ниже нормального	23	12	9	7	5	3
низший	22	11	8	6	4	2
простаивающий	16	1	1	1	1	1

Ограниченный буфер

- Имеется общий буфер ограниченной длины. Процесс-производитель добавляет в него сгенерированные элементы, процесс-потребитель использует и удаляет использованные элементы. Добавим в представление ограниченного буфера переменную counter,которую увеличивает процесс-производитель, добавляя очередной элемент к буферу, и уменьшает процесс-потребитель, используя и удаляя элемент из буфера.
- **Общие данные**

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

Ограниченный буфер

- **Процесс-производитель**

```
item nextProduced;

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

Ограниченный буфер

- **Процесс-потребитель**

```
item nextConsumed;

while (1) {
  while (counter == 0)
        ; /* do nothing */
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  counter--;
}
```

Ограниченный буфер

- **Оператор “count++” может быть реализован на языке ассемблерного уровня как:**

  register1 = counter
  register1 = register1 + 1
  counter = register1

- **Оператор “count—” может быть реализован как:**

  register2 = counter
  register2 = register2 – 1
  counter = register2

- Проблема в том, что если и производитель, и потребитель пытаются изменить переменную counter одновременно, то указанные ассемблерные операторы тоже должны быть **выполнены совместно (interleaved)**.

Ограниченный буфер

- **Предположим, counter вначале равно 5. Исполнение процессов в совместном режиме (interleaving) приводит к следующему:**

  producer: register1 = counter (*register1 = 5*)
  producer: register1 = register1 + 1 (*register1 = 6*)
  consumer: register2 = counter (*register2 = 5*)
  consumer: register2 = register2 – 1 (*register2 = 4*)
  producer: counter = register1 (*counter = 6*)
  consumer: counter = register2 (*counter = 4*)
- **Значение counter может оказаться равным 4 или 6, в то время как правильное значение counter равно 5.**

- Ситуация, при которой взаимодействующие процессы могут параллельно (одновременно) обращаться к общим данным, называется **конкуренцией за общие данные (race condition)**.Для предотвращения подобных ситуаций процессы следует **синхронизировать**.

Синхронизация процессов по критическим секциям

Рассмотрим проанализированную проблему в общем виде. Пусть имеются **n** параллельных процессов, каждый из которых может обратиться к общим для них данным. Назовем **критической секцией** фрагмент кода каждого процесса, в котором происходит обращение к общим данным.
Проблема **синхронизации процессов по критическим секциям** заключается в том, чтобы обеспечить следующий режим выполнения: если один процесс вошел в свою критическую секцию, то до ее завершения никакой другой процесс не смог бы одновременно войти в свою критическую секцию.

Можно показать, что для решения проблемы критической секции необходимо и достаточно выполнение следующих трех условий:

- **Взаимное исключение**. Если некоторый процесс исполняет свою критическую секцию, то никакой другой процесс не должен в этот момент исполнять свою.
- **Прогресс**.Если в данный момент нет процессов, исполняющих критическую секцию, но есть несколько процессов, желающих начать исполнение критической секции, то выбор системой процесса, которому будет разрешен запуск критической секции, не может продолжаться бесконечно.
- **Ограниченное ожидание**.В системе должно существовать ограничение на число раз, которое процессам разрешено входить в свои критические секции, начиная от момента, когда некоторый процесс сделал запрос о входе в критическую секцию, и до момента, когда этот запрос удовлетворен.

  При этом предполагается, что каждый процесс исполняется с ненулевой скоростью, но не делается никаких предположений о соотношении скоростей процессов.

Первоначальные попытки решения проблемы

- **Есть только два процесса, P_0 и P_1**
- **Общая структура процесса P_i:**

```
do {
        entry section
                critical section
        exit section
                remainder section
} while (1);
```

- **Процессы могут использовать общие переменные для синхронизации своих действий.**

Алгоритм 1

- **Общие переменные:**
  - **shared int turn;**
    первоначально turn = 0
  - **turn == i ⇒ процесс P_i может войти в критическую секцию**
- **Процесс P_i :**

```
do {
        while (turn != i) ;
                critical section
        turn = j;
                remainder section
} while (1);
```

- **Удовлетворяет принципу “взаимное исключение”, но не принципу “прогресс”:** алгоритм не предпринимает никаких мер, чтобы огран время выбора проца, желающего начать критическую секцию. Причина в следующем: алгоритм не хранит информацию о том, какие процессы желают войти в свои критические секции.

Алгоритм 2

- **Общие переменные**
  - **boolean flag[2];**
    первоначально flag [0] = flag [1] = false.
  - **flag [i] == true ⇒ P_i готов войти в критическую секцию**
- **Процесс P_i :**

```
do {
        flag[i] := true;
        while (flag[j]) ;                          critical section
        flag [i] = false;
        remainder section
      } while (1);
```

- **Удовлетв принципу “взаимное исключение”, но не принципу “прогресс”** алгоритм не различает информацию о том, что процесс еще только готов войти в свою критическую секцию, и о том, что он в нее уже вошел.

Алгоритм 3 (Петерсона, 1981)

- **Объединяет общие переменные алгоритмов 1 и 2.**
- **Процесс P_i :**

```
do {
        flag [i]:= true;
        turn = j;
        while (flag [j] and turn = j) ;
        critical section
        flag [i] = false;
        remainder section
} while (1);
```

- **Удовлетворяет всем трем принципам и решает проблему взаимного исключения.** перед входом в критическую секцию процесс сначала заявляет о своем намерении в нее войти, но затем пытается предоставить право на вход в критическую секцию **другому** процессу и только после того, как другой процесс ее выполнил и больше не желает в нее войти, входит сам в свою критическую секцию.

Алгоритм булочной (bakery algorithm)

алгоритм как бы воспроизводит стратегию автомата в (американской) булочной, где каждому клиенту присваивается его номер в очереди.

- **Обозначения: < ≡ лексикографический порядок**
- **(*a,b*) < (*c,d*) если *a* < *c* or if *a* = *c* and *b* < *d***
  - **max (a_0,…, a_{n-1})- число *k*, такое, что *k* ≥ a_i for *i* - 0, …, *n* – 1**
- **Общие данные:**

  boolean choosing[n];

  int number[n];

Структуры данных инициализируются, соответственно, false и 0

Алгоритм булочной

```
do {
 choosing[i] = true;
number [i] = max (number[0], number[1], …, number[n-1]) + 1;
 choosing[i] = false;
 for (j = 0; j < n; j++) {
     while choosing[j];
     while ((number[j] != 0) && (number[j] < number[i]));
 }
     критическая секция
     number [i] = 0;
     остальная часть кода
} while (1)
```

Аппаратная поддержка синхронизации

Атомарная операция проверки и модификации значения переменной

- TestAndSet, которая атомарно выполняет считывание и запоминание значения переменной, затем изменяет его на заданное значение, но в результате выдает первоначальное значение переменной.

.

```
boolean TestAndSet(boolean &target) {
        boolean rv = target;
        target = true;
        return rv;
}
```

Взаимное исключение с помощью TestAndSet

- Общие данные:
  boolean lock = false;

- Процесс P_i
  do {
  while (TestAndSet(lock)) ;
  critical section
  lock = false;
  remainder section
  } while (1)
- Значение переменной lock, равное true,означает, что вход в критическую секцию заблокирован. Каждый процесс ждет, пока он не разблокируется, затем, в свою очередь, выполняет блокировку и входит в критическую секцию. При ее завершении процесс разблокирует критическую секцию присваиванием lock значения false.

Аппаратное решение для синхронизации

- Атомарная перестановка значений двух переменных.

  Другое распространенное аппаратное решение для синхронизации – атомарная операция Swap, выполняющая перестановку значений двух переменных:

```
void Swap (boolean * a, boolean * b) {
    boolean temp = * a;
    * a = * b;
    * b = temp;
}
```

Взаимное исключение с помощью Swap

- Общие данные (инициализируемые **false**):
  boolean lock;
  boolean waiting[n];

- Процесс P_i
  do {
  key = true;
  while (key == true)
  Swap(&lock, &key);
  critical section
  lock = false;
  remainder section
  } while (1)

При данной реализации, условием ожидания процесса перед входом в критическую секцию является условия (key == true),которое фактически означает то же, что и в предыдущей реализации, - закрытое состояние блокировщика, т.е., то, что другой процесс находится в своей критической секции. Когда критическая секция освободится (освобождение осуществляется присваиванием lock = false после завершения критической секции в исполнившем ее процессе), ее начнет исполнять текущий процесс.

Общие семафоры – counting semaphores (по Э. Дейкстре)

- Семафо́р — объект, позволяющий войти в заданный участок кода не более чем *n* потокам.

Семафор — это объект, с которым можно выполнить три операции.

- init(n):счётчик := n
- enter():ждать пока счётчик станет больше 0; после этого уменьшить счётчик на единицу.
- leave():увеличить счётчик на единицу.

- Средство синхронизации, не требующее активного ожидания.
- (Общий) семафор *S* – целая переменная
- Может использоваться только для двух атомарных операций:

wait (*S*):

```
while S≤ 0 do no-op;
    S--;
```

signal (*S*):

```
S++;
```

Критическая секция для N процессов

- Общие данные:
  semaphore mutex; //initially *mutex* = 1

- Процесс *Pi:*

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

Реализация семафора

- Семафор, по существу, является структурой из двух полей – целого значения и указателя на список ждущих процессов::

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

- Предполагаем наличие двух простейших операций:
  - **block** – задерживает исполнение процесса, исполнившего данную операцию.
  - **wakeup(*P*)** возобновляет исполнение приостановленного процесса **P**.

Реализация

- Определим семафорные операции следующим образом:

***wait*(*S*):**

```
S.value--;
if (S.value < 0) {
    добавить  текущий процесс к S.L;
    block;
}
```

***signal*(*S*):**

```
S.value++;
if (S.value <= 0) {
    удалить процесс P из S.L;
    wakeup(P);
}
```

Семафоры как общее средство синхронизации

- Исполнить действие *B* в процессе P_j только после того, как действие *A* исполнено в процессе P_i
- Использовать семафор *flag* , инициализированный 0
- Код:

P_i	P_j
A	*wait*(*flag*)
signal(*flag*)	*B*

- **Общие и двоичные семафоры**

Из рассмотренного ясно, что имеется два вида семафоров: **общий** - целая переменная с теоретически неограниченным значением - и **двоичный** - целая переменная, значениями которой могут быть только 0 или 1. Преимуществом двоичного семафора является его возможная более простая аппаратная реализация. Например, в системах "Эльбрус" и Burroughs 5000 реализованы команды атомарного семафорного считывания с проверкой**семафорного бита**.

Реализация общего семафора S с помощью двоичных семафоров

- Структуры данных:

  binary-semaphore S1, S2;

  int C:
- Инициализация:

  S1 = 1

  S2 = 0

  C = начальное значение общего семафора **S**

В данной реализации семафор S1 используется для взаимного исключения доступа к общей целой переменной C. Семафор S2 используется для хранения очереди ждущих процессов в случае, если общий семафор переходит в закрытое состояние.

Реализация операций над семафором S

- Операция *wait:*

```
wait(S1);
C--;
if (C < 0) {
        signal(S1);
        wait(S2);
}
signal(S1);
```

- Операция *signal*:

```
wait(S1);
        C ++;
        if (C <= 0)
                signal(S2);
        else
                signal(S1);
```

Задача “ограниченный буфер”

- **ограниченный буфер**:имеются процесс-производитель и процесс-потребитель, взаимодействующие с помощью циклического буфера ограниченной длины; производитель генерирует элементы информации и записывает в буфер; потребитель использует информационные элементы из буфера и удаляет их.
- Общие данные:

  semaphore full, empty, mutex;

  Начальные значения:

  full = 0, empty = n, mutex = 1

Процесс-производитель ограниченного буфера

```
do {
        …
    сгенерировать элемент в nextp
        …
    wait(empty);
    wait(mutex);
        …
    добавить nextp к буферу
        …
    signal(mutex);
    signal(full);
} while (1);
```

Процесс-потребитель ограниченного буфера

```
do {
        wait(full)
        wait(mutex);
                …
        взять (и удалить) элемент из буфера в nextc
                …
        signal(mutex);
        signal(empty);
                …
        использовать элемент из nextc
                …
} while (1);
```

- Поясним использование семафоров.
  Семафор mutex используется "симметрично"; над ним выполняется пара операций: wait … signal – семафорные скобки.
  Его роль – чисто взаимное исключение критических секций.

Задача “читатели-писатели”

- Суть задачи **читатели-писатели** в следующем: имеется общий ресурс и две группы процессов: **читатели** (которые могут только читать ресурс) и **писатели** (которые изменяют ресурс). В каждый момент работать с ресурсом может сразу несколько читателей (когда ресурс не изменяется писателями), но не более одного писателя. Необходимо синхронизировать их действия над ресурсом и обеспечить взаимное исключение соответствующих критических секций.
- Общие данные:

  semaphore mutex, wrt;

  Начальные значения:

  mutex = 1, wrt = 1, readcount = 0

Процесс-писатель

wait(wrt); …

выполняется запись **…**

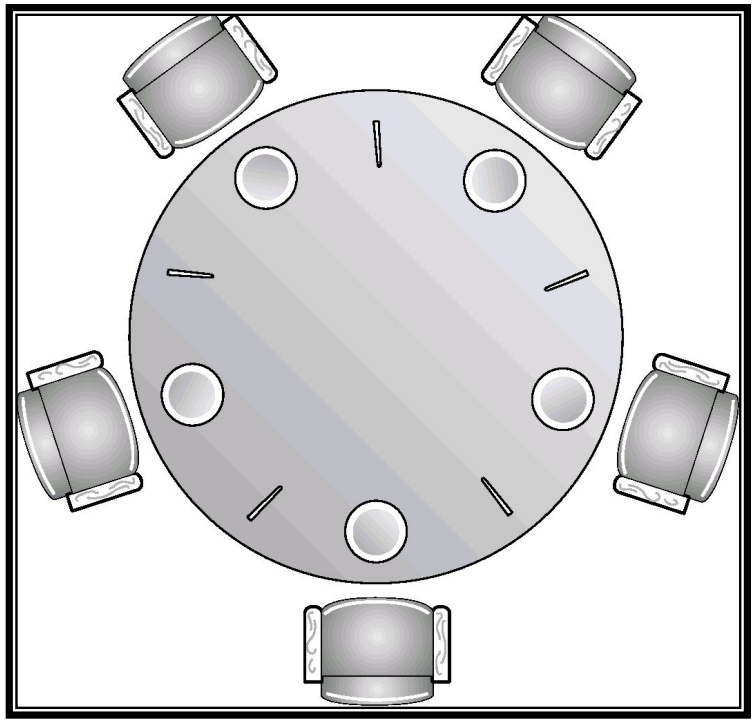
signal(wrt);

- Процесс-читатель, во-первых, должен увеличить значение readcount, причем обеспечить взаимное исключение для действий над readcount с помощью семафора mutex.Далее, если процесс является первым читателем, он должен закрыть семафор wrt, чтобы исключить одновременное с чтением изменение ресурса писателями. По окончании чтения ресурса, читатель в аналогичном стиле вновь уменьшает readcount. Если при этом оно обнуляется (т.е. это последний на данный момент читатель), то читатель открывает семафор wrt, сигнализируя писателям, что они могут изменять ресурс.

Процесс-читатель

```
    wait(mutex);
    readcount++;
    if (readcount == 1){
        wait(rt);
}
    signal(mutex);
            …
        выполняется чтение
            …
    wait(mutex);
    readcount--;
    if (readcount == 0){
        signal(wrt);
    signal(mutex):
}
```

Задача “обедающие философы”



- **Общие данные** При решении задачи будем использовать массив семафоров chopstick, описывающий текущее состояние палочек:chopstick[i] закрыт, если палочка занята, открыт – если свободна:

  semaphore chopstick[5] ={ 1, 1, 1, 1, 1};**;**

  Первоначально все значения равны 1

Задача “обедающие философы”

Имеется круглый стол, за которым сидят пять философов (впрочем, их число принципиального значения не имеет – для другого числа философов решение будет аналогичным). Перед каждым из них лежит тарелка с едой, слева и справа от каждого – две китайские палочки. Философ может находиться в трех состояниях: **проголодаться, думать** и **обедать**. На голодный желудок философ думать не может. Но начать обедать он может, только если обе палочки слева и справа от него свободны. Требуется синхронизировать действия философов. В данном случае общим ресурсом являются палочки. Иллюстрацией условий задачи является

- Философ *i*:

```
do {
        wait(chopstick[i]);
        wait(chopstick[(i+1) % 5]);
                …
                dine
                …
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
                …
                think
                …
} while (1);
```

Пример: ограниченный буфер

- Общие данные:

```
struct buffer {
        int pool[n];
        int count, in, out;
        }
```

- **Критические области (critical regions) –** более высокоуровневая и надежная конструкция для синхронизации, чем семафоры. Общий ресурс описывается в виде особого описания переменной:
- v: shared T
- Доступ к переменной v возможен только с помощью специальной конструкции:
- region v when B do S
- где v – общий ресурс; B – булевское условие, S – оператор (содержащий действия над v).
- Семантика данной конструкции следующая. Пока B ложно, процесс, ее исполняющий, должен ждать. Когда Bстановится истинным, процесс получает доступ к ресурсу v и выполняет над ним операции S. Пока исполняется оператор S, больше ни один процесс не имеет доступа к переменной v.

Процесс-производитель

- Процесс-производитель добавляет **nextp** к общему буферу

```
region buffer when (count < n) {
    pool[in] = nextp;
    in:= (in+1) % n;
    count++;
}
```

проверка переполнения буфера выполняется при входе в конструкцию **region**, и потребитель ждет, пока в буфере освободится хотя бы один элемент.

Процесс-потребитель

- Процесс-потребитель удаляет элемент из буфера и запоминает его в **nextc**

```
 region buffer when (count > 0) {                    nextc =
pool[out];
        out = (out+1) % n;
        count--;
 }
```

Реализация оператора region *x* when *B* do *S*

- Свяжем с общей переменной *x* следующие переменные:

  semaphore mutex, first-delay, second-delay;
  int first-count, second-count;
- Взаимное исключение доступа к критической секции обеспечивается семафором **mutex**.
- Если процесс не может войти в критическую секцию, т.к. булевское выражение **B** ложно, он ждет на семафоре **first-delay**; затем он “перевешивается” на семафор **second-delay,** до тех пор, пока ему не будет разрешено вновь вычислить *B*.

Мониторы (C. A. R. Hoare)

- **Высокоуровневая конструкция для синхронизации, которая позволяет синхронизировать доступ к абстрактному типу данных**.

```
monitor monitor-name
{
    описания общих переменных
    procedure body P1 (…) {
        . . .
    }
    procedure body P2 (…) {
        . . .
    }
    procedure body Pn (…) {
        . . .
    }
    {
        код инициализации
    }
}
```

Мониторы: условные переменные

- **Для реализации ожидания процесса внутри монитора, вводятся условные переменные:**

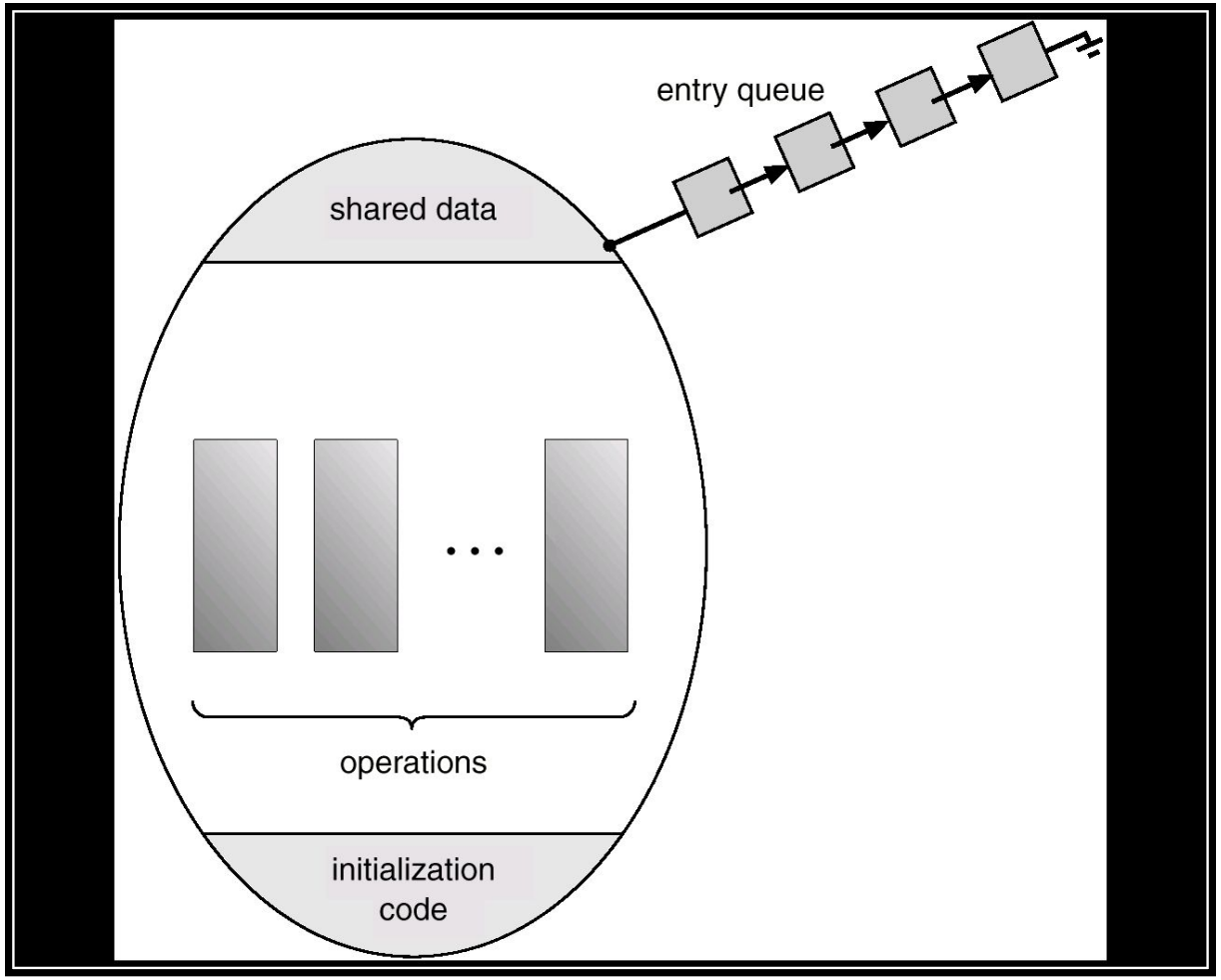
  condition x, y;
- **Условные переменные могут использоваться только в операциях wait и signal.**
  - **Операция:**

    x.wait();

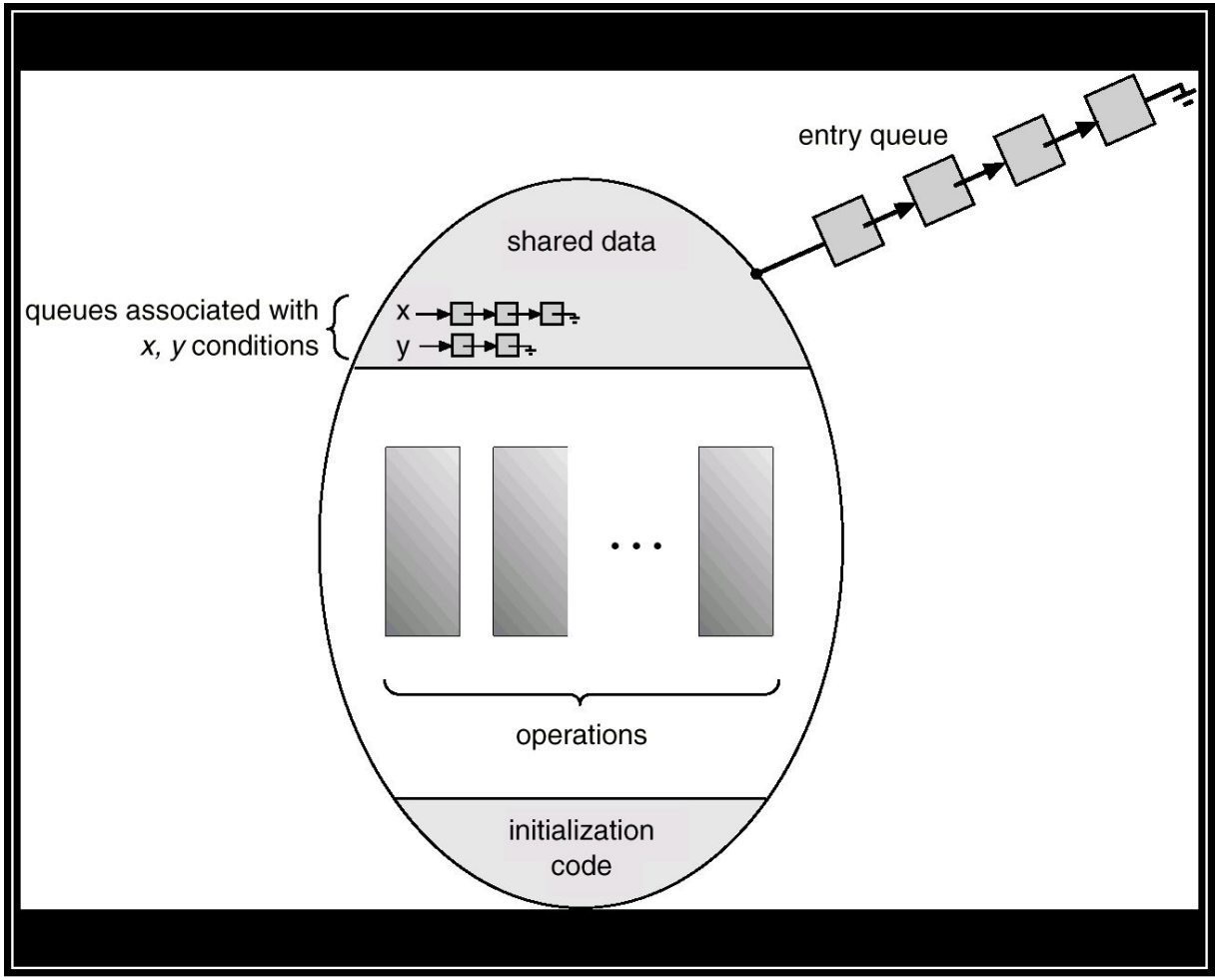
    означает, что выполнивший ее процесс задерживается до того момента, пока другой процесс не выполнит операцию:

    x.signal();
  - **Операция x.signal возобновляет ровно один приостановленный процесс. Если приостановленных процессов нет, эта операция не выполняет никаких действий.**
- **Монитор является многовходовым модулем особого рода. Он содержит описания общих для нескольких параллельных процессов данных и операций над этими данными в виде процедур P1, …, Pn. Пользователи монитора – параллельные процессы – имеют доступ к описанным в нем общим данным только через его операции, причем в каждый момент времени не более чем один процесс может выполнять какую-либо операцию монитора; остальные процессы, желающие выполнить операцию монитора, должны ждать, пока первый процесс закончит выполнять мониторную операцию**.

Схематическое представление монитора



Монитор с условными переменными



Синхронизации

- **Синхронизация в ОС Solaris**
- Система Solaris предоставляет разнообразные виды блокировщиков для поддержки многозадачности, многопоточности (включая потоки реального времени) и мультипроцессирования. Используются адаптивные мьютексы (adaptive mutexes) – эффективное средство синхронизации доступа к данным при их обработке короткими сегментами кода. Для более длинных сегментов кода используются условные переменные и блокировщики читателей-писателей (reader-writer locks; rwlocks).Для синхронизации потоков используются "вертушки" (turnstiles) – синхронизирующие примитивы, которые позволяют использовать либо adaptive mutex, либо rwlock.
- **Синхронизация в Windows 2000**
- Для защиты доступа к данным на однопроцессорных системах используются маски прерываний. Для многопроцессорных систем используются spinlocks (" вертящиеся замки. В системе реализованы также объекты-диспетчеры, которые могут функционировать как мьютексы и как семафоры. Объекты-диспетчеры генерируют события, семантика которых аналогична семантике условной переменной.

Пример: обедающие философы

```
monitor dp
{
  enum {thinking, hungry, eating} state[5];
  condition self[5];
  void pickup(int i)      // following slides
  void putdown(int i)   // following slides
  void test(int i)              // following slides
  void init() {
        for (int i = 0; i < 5; i++)
              state[i] = thinking;
  }
}
```

Обедающие философы: реализация операций pickup и putdown

```
void pickup(int i) {
  state[i] = hungry;
  test[i];
  if (state[i] != eating)
        self[i].wait();
}

void putdown(int i) {
  state[i] = thinking;
  // test left and right neighbors
  test((i+4) % 5);
  test((i+1) % 5);
}
```

Обедающие философы: реализация операции test

```
void test(int i) {
  if ( (state[(i + 4) % 5] != eating) &&
    (state[i] == hungry) &&
    (state[(i + 1) % 5] != eating)) {
        state[i] = eating;
        self[i].signal();
  }
}
```

Реализация мониторов с помощью семафоров

Используем семафоры mutex – для взаимного исключения процессов, next –для реализации очереди входа в монитор; переменную next-count – счетчик процессов в очереди на вход:

- Переменные

  semaphore mutex; // (initially = 1)
  semaphore next; // (initially = 0)
  int next-count = 0;

- Каждая внешняя процедура ***F*** заменяется на:

```
wait(mutex);
    …
 тело F;
    …
if (next-count > 0)
      signal(next)
else
      signal(mutex);
```

- Обеспечивается взаимное исключение внутри монитора.

Реализация мониторов

- Для каждой условной переменной ***x***:

  semaphore x-sem; // (initially = 0)

  int x-count = 0;

- Реализация операции **x.wait**:

```
x-count++;
if (next-count > 0)
        signal(next);
else
        signal(mutex);
wait(x-sem);
x-count--;
```

Реализация мониторов

- Реализация операции **x.signal**:

```
if (x-count > 0) {
        next-count++;
        signal(x-sem);
        wait(next);
        next-count--;
}
```

Таким образом, обеспечивается, что процесс, освобожденный из очереди к условной переменной, помещается во входную очередь монитора.

Дополнительная операция над монитором, обеспечивающая организацию очереди к условной переменной по приоритетам, - x.wait(c),где c – целочисленный параметр, играющий роль приоритета. При выполнении операцииsignal первым будет освобожден из очереди процесс с меньшим значением приоритета.

При реализации монитора необходимо проверять следующие условия:

- процессы должны выполнять вызовы операций монитора в правильной последовательности, своевременно вызывая все семафорные операции;
- никакой процесс не пытается обратиться к общим данным непосредственно, минуя протокол взаимодействия с монитором.

Тупики

Модель системы

Для описания и исследования подобных ситуаций введем формальную модель системы в общем виде. С помощью модели будем представлять информацию о запросах процессов к ресурсам, о фактическом владении процессов ресурсами и об освобождении ресурсов.

Пусть в системе имеется **m** видов ресурсов (например, процессор, память, устройства ввода-вывода). Будем обозначать типы ресурсов в системе **R1, R2, … Rm.** Пусть каждый тип ресурса **Ri** имеет **Wi** экземпляров.

Каждый процесс может использовать ресурс одним из следующих способов:

- **запрос (request)**
- **использование (use)**
- **освобождение (release).**

Тупик может возникнуть, если одновременно выполняются следующие четыре условия:

- **взаимное исключение**: только один процесс в каждый момент времени может получить доступ к ресурсу;
- **удержание и ожидание**: процесс, удерживающий один ресурс, ожидает приобретения других ресурсов, которыми обладают другие процессы;
- **отсутствие прерываний:** процесс может освободить ресурс только добровольно, когда завершит свою работу;
- **циклическое ожидание**: существует множество **{P0, P1, … P0},** такое, что **P0** ожидает ресурса, которым обладает **P1; P1** ожидает ресурса, которым обладает **P2 … Pn** ожидает ресурса, которым обладает **P0**.

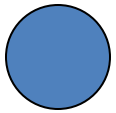
Граф распределения ресурсов

- Множество вершин *V* и множество дуг *E*.
- *V* подразделяется на два типа вершин:
  - $P = \{P_1, P_2, \ldots, P_n\}$, множество всех процессов в системе.
  - $R = \{R_1, R_2, \ldots, R_m\}$, множество всех ресурсов в системе.
- Дуга типа “запрос” (request edge) – направленная дуга $P_i \rightarrow R_j$
- Дуга типа “присваивание” (assignment edge) – направленная дуга $R_j \rightarrow P_i$

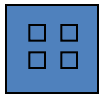
Смысл различных направленностей дуг в следующем. Если процесс претендует на какой-либо ресурс, то дуга проводится из вершины-процесса в вершину-ресурс. Когда же конкретная единица ресурса уже выделена какому-либо конкретному процессу, то дуга, в знак этой принадлежности, и проводится из вершины-ресурса в вершину процесс.

Граф распределения ресурсов (продолжение)

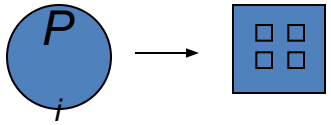
- **Процесс**



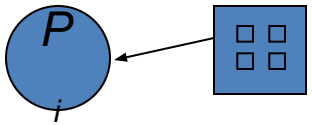
- **Тип ресурса, имеющий 4 экземпляра**



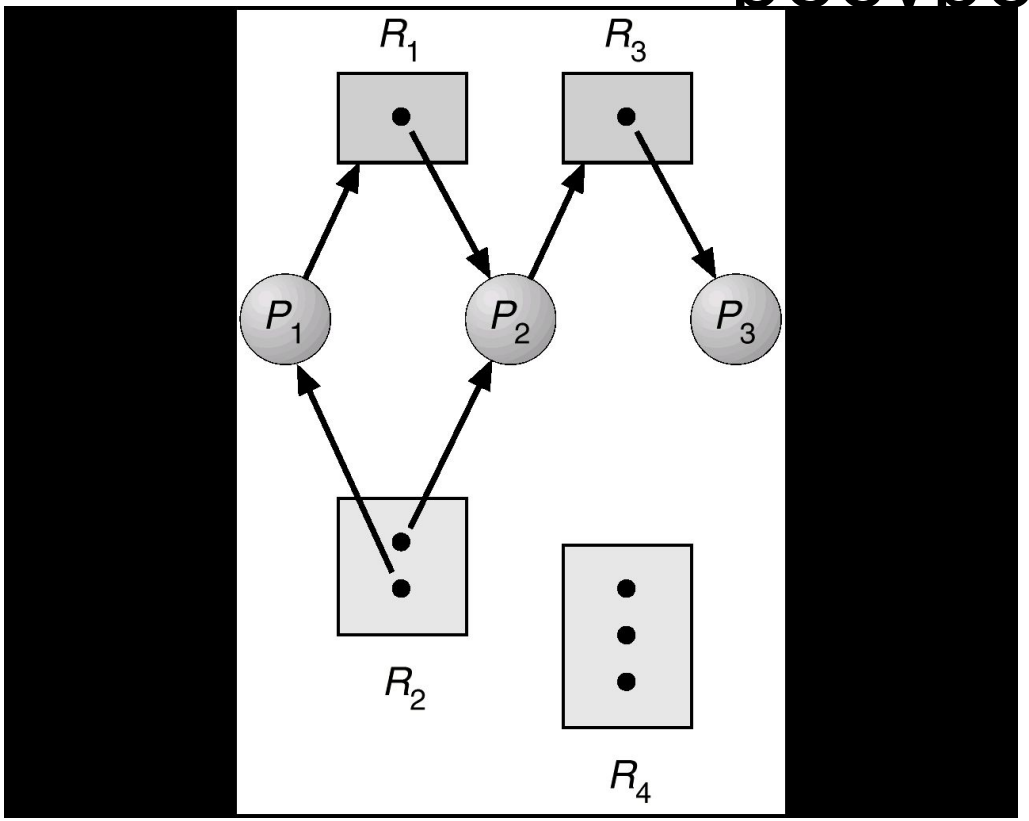
- **P_i запрашивает экземпляр ресурса R_j**



- **P_i удерживает экземпляр ресурса R_j**

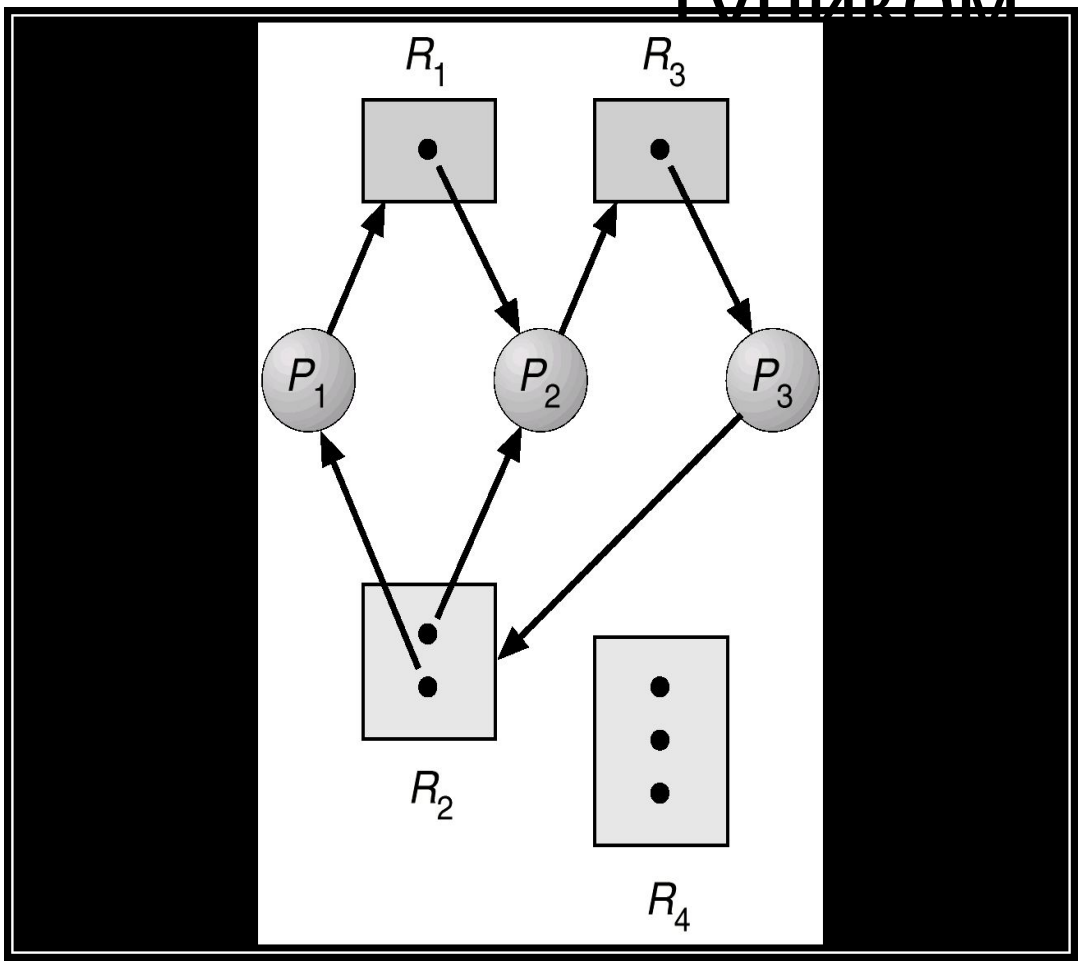


Пример графа распределения ресурсов



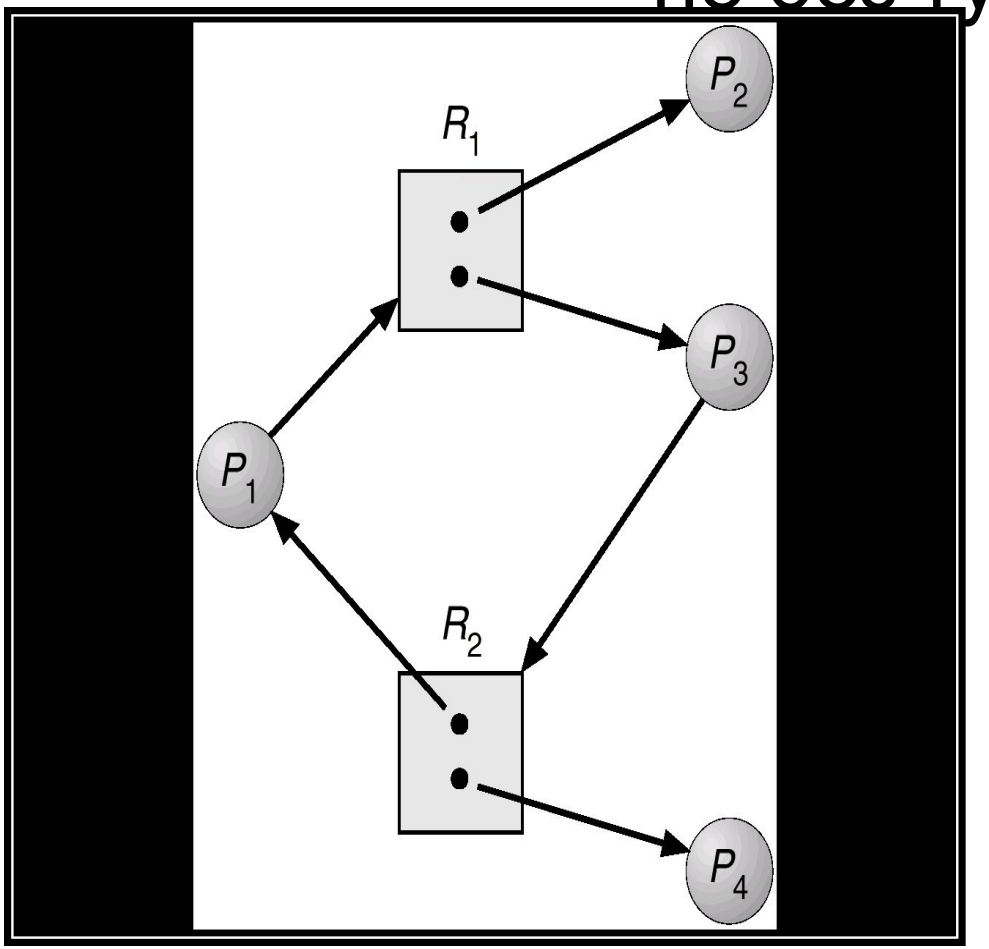
Данный граф изображает систему с тремя процессами и четырьмя видами ресурсов: ресурсы видов 1 и 3 имеют по одному экземпляру, ресурс вида 2 – два экземпляра, ресурс вида 4 – три экземпляра. Процесс 1 претендует на ресурс 1, который занят процессом 2. Процесс 2 претендует на ресурс 3, который занят процессом 3. Две единицы ресурса 2 отданы процессам 1 и 2. Ресурс 4 не распределялся (все три единицы свободны).

Граф распределения ресурсов с тупиком



Имеется ситуация циклического ожидания между процессами 1, 2 и 3. Процесс 1 претендует на ресурс, которым владеет процесс 2. Процесс 2 претендует на ресурс, которым владеет процесс 3. Процесс 3 претендует на ресурс, одна единица которого отдана процессу 1, а вторая – процессу 2.

Граф распределения ресурсов с циклом, но без тупика



В данном случае (имеется четыре процесса и два вида ресурсов. В цикле участвуют вершины-процессы 1 и 3. Однако, благодаря тому, что каждого ресурса имеется по две единицы, тупика удается избежать: процесс 1, ожидающий ресурса 1, сможет его получить, когда завершится процесс 2 (а не процесс 1), обладающий одной единицей данного ресурса и не входящий в цикл ожидания. Аналогично, процесс 3, претендующий на ресурс 2, сможет его получить после его освобождения процессом 4 (а не 1).

- Если граф распределения ресурсов не содержит циклов, то в системе тупиков нет;
- Если граф распределения ресурсов содержит цикл, то возможно два случая:
  - Если ресурсов каждого вида имеется только по одному экземпляру, то имеет место тупик;
  - Если ресурсов по несколько экземпляров, то тупик возможен.
- **Методы обработки тупиков**
- Теоретически возможны следующие методы обработки тупиков:
- Убедиться в том, что система никогда не войдет в состояние тупика;
- Допустить, чтобы система могла входить в состояние тупика, но предусмотреть возможность **восстановления**после тупика.
- К сожалению, на практике во многих ОС (включая UNIX) используется и третий "метод" борьбы с тупиками: проблема тупиков игнорируется, но авторы ОС без каких-либо обоснований претендуют на то, что в системе тупики невозможны.

Предотвращение тупиков

- Основная идея – ограничить методы запросов ресурсов со стороны процессов.
- Чтобы ограничить возможность взаимного исключения владения ресурсами (первое условие тупика), необходимо заметить, что оно требуется не для всех ресурсов. Для разделяемых ресурсов (например, массивов констант, кодов, файлов) оно не требуется.

- Чтобы ограничить возможность удержания и ожидания (второе условие тупика), можно потребовать, чтобы процесс, запрашивающий некоторый ресурс, не обладал бы больше никакими ресурсами. Альтернативным вариантом является требование, чтобы все процессы приобретали все необходимые им ресурсы до фактического начала их исполнения.

- Более разумной представляется стратегия перераспределения ресурсов при каждом ожидании процессом ресурса. Если процесс обладает некоторым ресурсом **A** и запрашивает другой ресурс **B**, который не может быть ему немедленно выделен, то процесс должен ждать. При этом ресурс **A,** занимаемый процессом, должен быть **немедленно освобожден.** Ресурс **A** добавляется к списку ресурсов, которые ожидает процесс. Процесс может быть возобновлен, только если ему могут быть выделены одновременно все старые ресурсы, которыми он обладал, и те новые ресурсы, которых он ожидает.

- Для предотвращения ситуации циклического ожидания самое простое решение – ввести упорядочение по номерам всех видов ресурсов и требовать, чтобы процесс запрашивал ресурсы только в порядке возрастания их номеров.

Избежание тупиков

- Методы избежания тупиков требуют, чтобы система обладала дополнительной априорной информацией о процессе и его потребностях в ресурсах с момента ввода каждого процесса в систему.
- Наиболее простая и полезная модель требует, чтобы каждый процесс при вводе в систему указывал максимальный объем ресурсов каждого типа, которые могут ему понадобиться. Данный подход был реализован даже в ранних ОС и носит название **паспорт задачи –** список максимальных потребностей процесса в ресурсах каждого типа – оперативной и внешней памяти, времени выполнения, листах печати и др.
- Алгоритм избежания тупиков должен анализировать состояние распределения ресурсов, чтобы убедиться, что никогда не может возникнуть ситуация циклического ожидания.
- Состояние распределения ресурсов описывается как объем доступных ресурсов, объем распределенных ресурсов и максимальные требования процессов.

Безопасное состояние системы

- **Б С** назовем такое состояние, перевод системы в которое не приведет к появлению тупиков.

  Общий принцип избежания тупиков состоит в следующем. Когда процесс запрашивает доступный ресурс, система должна определить, приведет ли немедленное выделение данного ресурса к безопасному состоянию системы.

  Система находится в **безопасном состоянии**, если существует безопасная последовательность, состоящая из всех процессов в системе.

  Безопасной последовательностью процессов называется последовательность процессов $<P_1, \dots P_n>$, такая, что для каждого процесса P_i ресурсы, которые он может еще запросить, могут быть выделены из текущих доступных ресурсов и ресурсов, удерживаемых процессами P_j , где $j < i$.

  Если последовательность процессов безопасна, то система может придерживаться следующей безопасной стратегии, с точки зрения распределения ресурсов и исполнения процессов:

  Если потребности процесса P_i в ресурсах не могут быть немедленно удовлетворены, то процесс может подождать, пока завершатся процессы P_j (где $j < i$), удерживающие требуемые ресурсы;

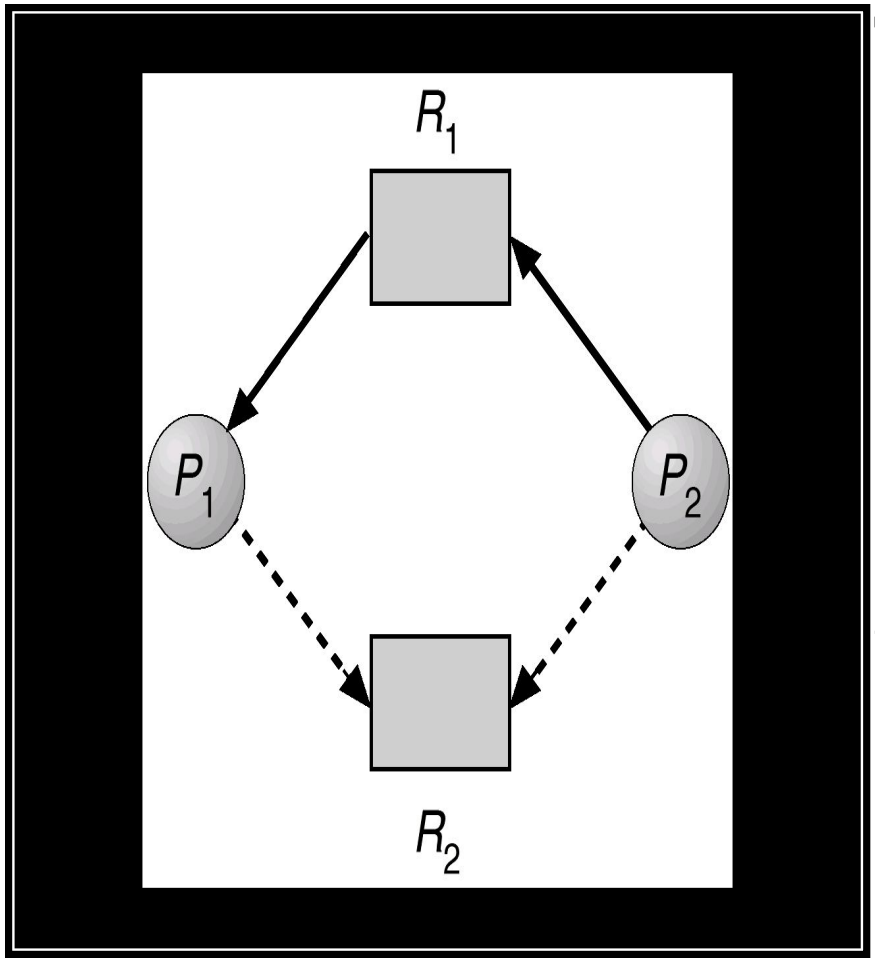
  Когда процессы P_j завершены, процесс P_i может получить требуемые ресурсы, выполниться, вернуть удерживаемые ресурсы и завершиться;

После завершения процесса P_i , процесс P_{i+1} может получить требуемые им ресурсы, и т.д.

Таким образом, справедливы следующие утверждения:

- Если система в безопасном состоянии, тупиков нет;
- Если системы в небезопасном состоянии, тупики возможны;
- Для того, чтобы избежать тупиков, необходимо проверять перед выделением ресурсов, что система никогда не придет в небезопасное состояние.

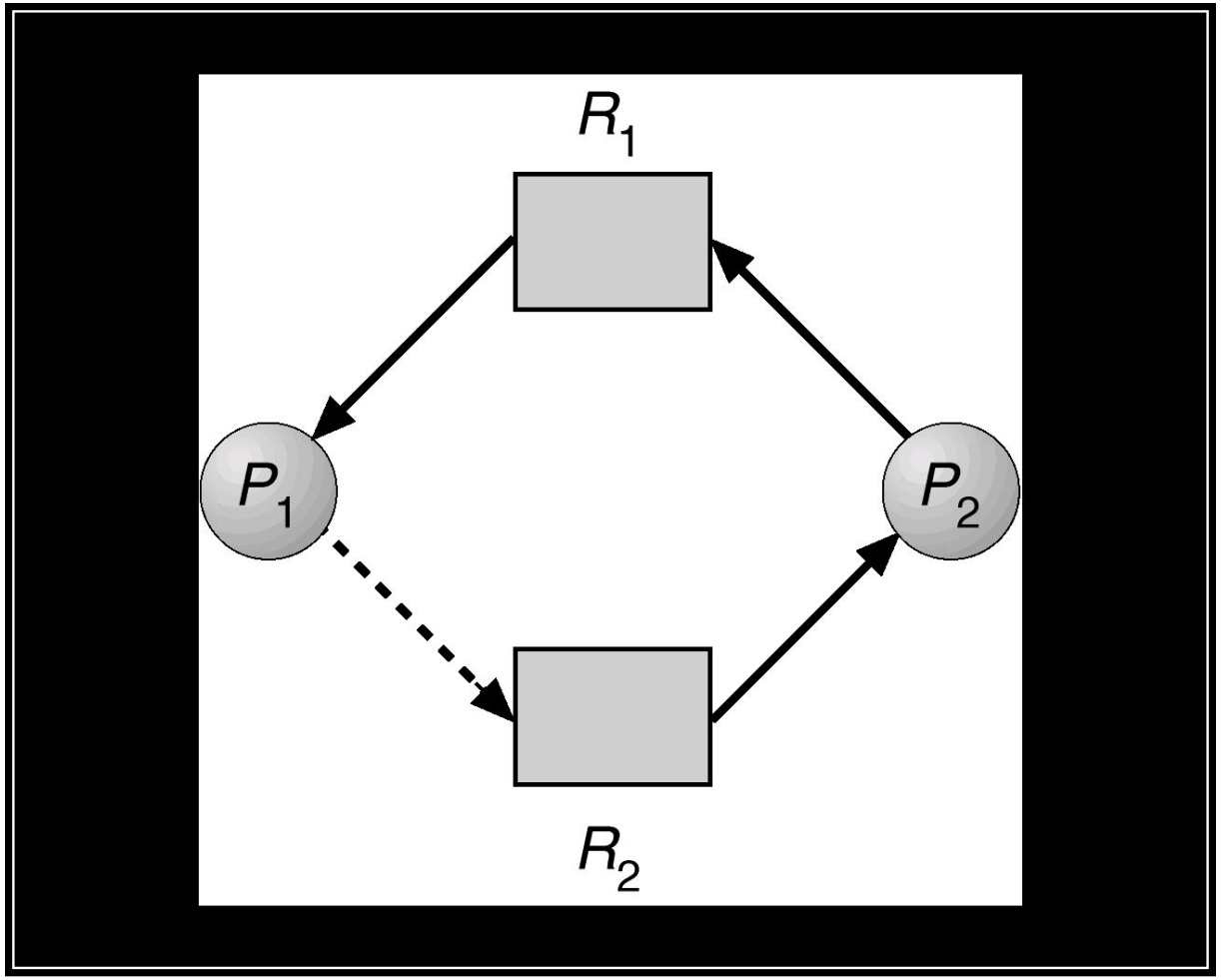
Граф распределения ресурсов для стратегии избежания тупиков



Для реализ стратегии ИТ к данному графу необходимо добавить информацию не только о фактических, но и о **возможных** в будущем запросах ресурсов со стороны процессов. Для этого, в дополнение к дугам запросов и присваиваний, введем в рассмотрение **дугу потребности (claim edge)**, кот ведет из вершины-процесса P_i в вершину-ресурс R_j, обозначается пунктирной линией и означ, что проц P_i может потреб ресурс R_j.Когда процесс фактически запрашивает данный ресурс, дуга потребности преобраз в дугу запроса (пунктирная линия-> сплошной).

Когда проц освобождает ресурс, дуга присваивания преобраз обратно в дугу потребности.Цель данной модификации графа – обеспечить, чтобы потребность в ресурсах была априорно известна системе.

Небезопасное состояние на графе распределения ресурсов



Алгоритм банкира

- Алгоритм банкира для безопасного распределения ресурсов операционной системой (с избежанием тупиков) был предложен Э. Дейкстрой и впервые реализован в операционной системе THE в конце 1960-х гг. Происхождение названия связано с тем, что поведение алгоритма напоминает осторожную стратегию банкира при проведении банковских операций. Принципы алгоритма банкира следующие.

  Каждый процесс должен априорно обозначить свои потребности в ресурсах по максимуму.

  Когда процесс запрашивает ресурс, ему, возможно придется подождать (выделение ресурсов по запросу не всегда может произойти немедленно).

  Когда процесс получает требуемые ресурсы, он должен их вернуть системе за ограниченный период времени.

  Структуры данных для алгоритма банкира

Пусть *n* = число процессов; *m* = число типов ресурсов

- ***Avaliable:*** **Вектор длины *m*. Если *available* [*j*] = *k*, до в данный момент доступны *k* экземпляров ресурса типа R_j .**
- ***Max:*** **Матрица *n* * *m*. Если *Max* [*i,j*] = *k*, то процесс P_i может запросить самое большее *k* экземпляров ресурса типа R_j.**
- ***Allocation:*** **Матрица *n* *m*. Если *Allocation*[*i,j*] = *k* , то процессу P_i в данный момент выделено *k* экземпляров ресурса типа R_j.**
- ***Need:*** **Матрица *n* * *m*. Если *Need*[*i,j*] = *k*, то P_i может потребоваться еще *k* экземпляров ресурса R_j для завершения своей работы.**

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j].$$

Алгоритм безопасности

1.Пусть *Work* и *Finish* – векторы длин *m* и *n*, соответственно. Инициализация:

Work = *Available*

Finish [*i*] = *false* для *i* = 1,2, …, *n.*

2. Находим *i* такое, что:

(a) *Finish* [*i*] = *false*

(b) *Need*[i] ≤ *Work*

Если такого *i* нет, переходим к шагу 4.

3. *Work* = *Work* + *Allocation[I]*

Finish[*i*] = *true*

Переход к шагу 2.

4. Если *Finish* [*i*] == true для всех *i*, то система в безопасном состоянии.

- Алгоритм строит безопасную последовательность номеров процессов i, если она существует. На каждом шаге, после обнаружения очередного элемента последовательности, алгоритм моделирует освобождение i - мпроцессом ресурсов после его завершения.
- На шаге 1 присваивание векторов выполняется поэлементно.
- На шаге 2, Need – матрица потребностей (n * m); Need[i] - строка матрицы, представляющая вектор потребностей (длины m) процесса i. Сравнение выполняется поэлементно, и его результат считается истинным, если соотношение выполнено для всех элементов векторов
- На шаге 3, Allocation [i] С помощью вектора Work моделируется освобождение ресурсов i – м процессом, после чего процессу присваивается признак завершаемости.

Алгоритм запроса ресурсов для процесса P_i

Request **= вектор запросов для процесса** P_i**. Если** $Request_i[j] = k$ **, то процесс** P_i **запрашивает** ***k*** **экземпляров ресурса типа** R_j.

1. **Если** $Request_i \leq Need_i$ **, перейти к шагу 2. Иначе сгенерировать исключительную ситуацию, т.к. Процесс превысил свои максимальные потребности.**
2. **Если** $Request_i \leq Available$**, перейти к шагу 3. Иначе процесс** P_i **должен ждать, так как ресурс недоступен.**
3. **Спланировать выделение ресурса процессу** P_i **, модифицируя состояние следующим образом:**

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_{i;;}$$

- ***Если состояние безопасно ⇒ ресурс выделяется*** P_i.
- ***Если состояние небезопасно ⇒*** P_i ***должен ждать; восстанавливается предыдущее состояние распределения ресурсов***

Пример (продолжение)

- **Состояние матрицы. Need определяется как (Max – Allocation).**

	Need
	A B C
P_0	**7 4 3**
P_1	**1 2 2**
P_2	**6 0 0**
P_3	**0 1 1**
P_4	**4 3 1**

- **Система в безопасном состоянии, т.к. $< P_1, P_3, P_4, P_2, P_0 >$ удовлетворяет критерию безопасности.**

Пример (продолжение). Запрос процесса $P_{1:}$ (1,0,2)

- **Проверяем, что Request ≤ Available, то есть, (1,0,2) ≤ (3,3,2) ⇒ *true.***

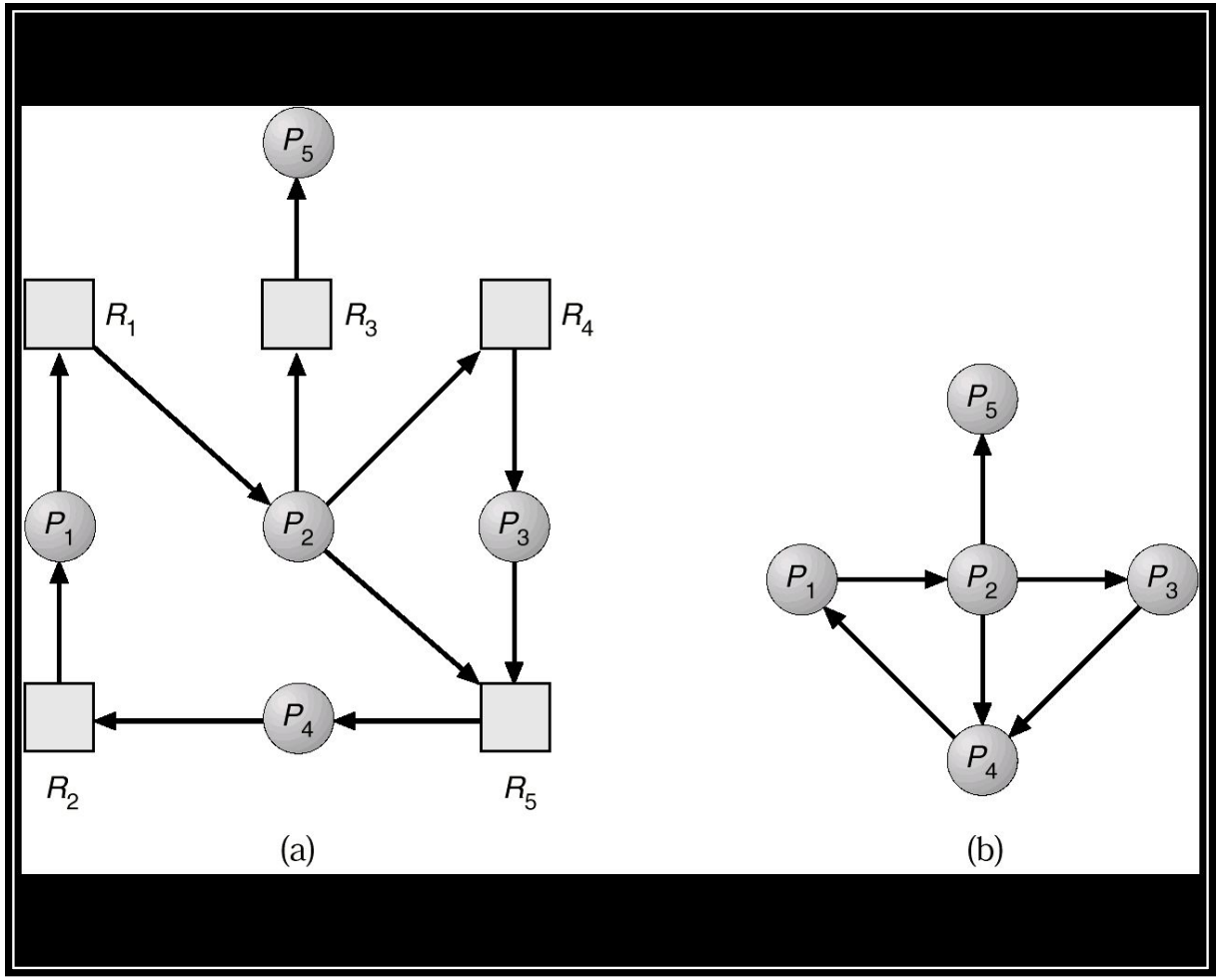
	Allocation	***Need***	***Available***
	A B C	***A B C***	***A B C***
P_0	**0 1 0**	**7 4 3**	**2 3 0**
P_1	**3 0 2**	**0 2 0**	
P_2	**3 0 1**	**6 0 0**	
P_3	**2 1 1**	**0 1 1**	
P_4	**0 0 2**	**4 3 1**	

- **Исполнение алгоритма безопасности показывает, что <P_1, P_3, P_4, P_0, P_2> удовлетворяет критерию безопасности.**

Случай, когда каждый тип ресурса имеет единственный экземпляр

- **Строим и поддерживаем *wait-for* граф**
  - **Вершины - процессы.**
  - **$P_i \rightarrow P_j$, если P_i ожидает P_j.**
- **Периодически вызываем алгоритм, который проверяет отсутствие циклов в этом графе.**
- **Алгоритм обнаружения цикла в графе требует $O(n^2)$ операций, где n – число вершин в графе.**

Граф распределения ресурсов и граф wait-for



Случай, когда ресурсы существуют в нескольких экземплярах для каждого типа

- ***Available:*** **Вектор длины *m*; указывает наличие ресурсов каждого типа.**

- ***Allocation:*** **Матрица *n* x *m* , определяющая число ресурсов каждого типа, выделенных каждому процессу.**

- ***Request:*** **Матрица *n* x *m* , задающая запросы для каждого процесса. Если *Request* [i_j] = *k*, то процесс P_i запрашивает (еще) *k* экземпляров ресурса типа R_j.**

Алгоритм обнаружения тупиков

Пусть *Work* и *Finish* – векторы длин *m* и *n*, соответственно.

1. Инициализация:

(a) *Work* = *Available*

(b) For *i* = 1,2, …, *n*, if $Allocation_i \neq 0$, then
***Finish*[i] = false; otherwise, *Finish*[i] = *true*.**

2. Найти индекс *i* такой, что:

(a) *Finish*[*i*] == *false*

(b) $Request_i \leq Work$

Если нет такого *i* , перейти к шагу 4.

3. $Work = Work + Allocation_i$
Finish*[*i*] = *true
перейти к шагу 2.

4. Если *Finish*[*i*] == false для некоторого *i*, $1 \leq i \leq n$, то система в состоянии тупика. Более того, если *Finish*[*i*] == *false*, то P_i – в состоянии тупика.

Алгоритм требует $O(m \times n^2)$ операций для определения того, находится ли система в тупиковом состоянии.

Алгоритм обнаружения: пример

- **Пусть имеются пять процессов P_0 - P_4 и три типа ресурсов A (7 экземпляров), *B* (2 экземпляра) и *C* (6 экземпляров).**
- **В момент времени T_0:**

	Распределение		***Запрос***		
	A	***B***	***C***	***A***	***B***	***C***
P_0	0	1	0	0	0	0	0	0
P_1	2	0	0	2	0	2		
P_2	3	0	3	0	0	0		
P_3	2	1	1	1	0	0		
P_4	0	0	2	0	0	2		

- **Последовательность <P_0, P_2, P_3, P_1, P_4> будет завершена *Finish*[*i*] = true для всех *i*.**

>системе следует использовать рассмотренный алгоритм обнаружения тупиков, зависит от того, как часто, по всей вероятности, будет иметь место тупик и сколько процессов будет необходимо откатить назад, чтобы выйти из тупика. Ответ на последний вопрос: **по одному процессу для каждого из не пересекающихся циклов**.

Алгоритм обнаружения: продолжение

- **P_2 запрашивает дополнительный ресурс типа *C*.**

	Запрос
	A B C
P_0	**0 0 0**
P_1	**2 0 1**
P_2	**0 0 1**
P_3	**1 0 0**
P_4	**0 0 2**

- **Состояние системы?**
  - **Может вновь запросить ресурсы, которыми обладает процесс P_0, но не имеет достаточно ресурсов, чтобы удовлетворить запросы друих процесов.**
  - **Имеет место тупик, в котором находятся процессы P_1, P_2, P_3 и P_4.**

Восстановление после тупика

Для выхода из тупика, очевидно, система должна прекратить все заблокированные процессы и освободить занимаемые ими ресурсы. Для более оптимального выполнения данного действия, система может прекращать на каждом шаге по одному процессу и после этого анализировать, ликвидирован ли тупик.

Важный вопрос – **в каком порядке** необходимо прекращать процессы? Существуют различные подходы:

- В порядке приоритетов процессов;
- В зависимости то того, насколько долго процесс уже выполняется и сколько времени осталось до его завершения;
- В зависимости от объема ресурсов, которые удерживал процесс;
- В зависимости от объема ресурсов, требуемого для завершения процесса;
- В зависимости от того, сколько всего процессов требуется прекратить;
- В зависимости от того, является ли процесс интерактивным или пакетным.

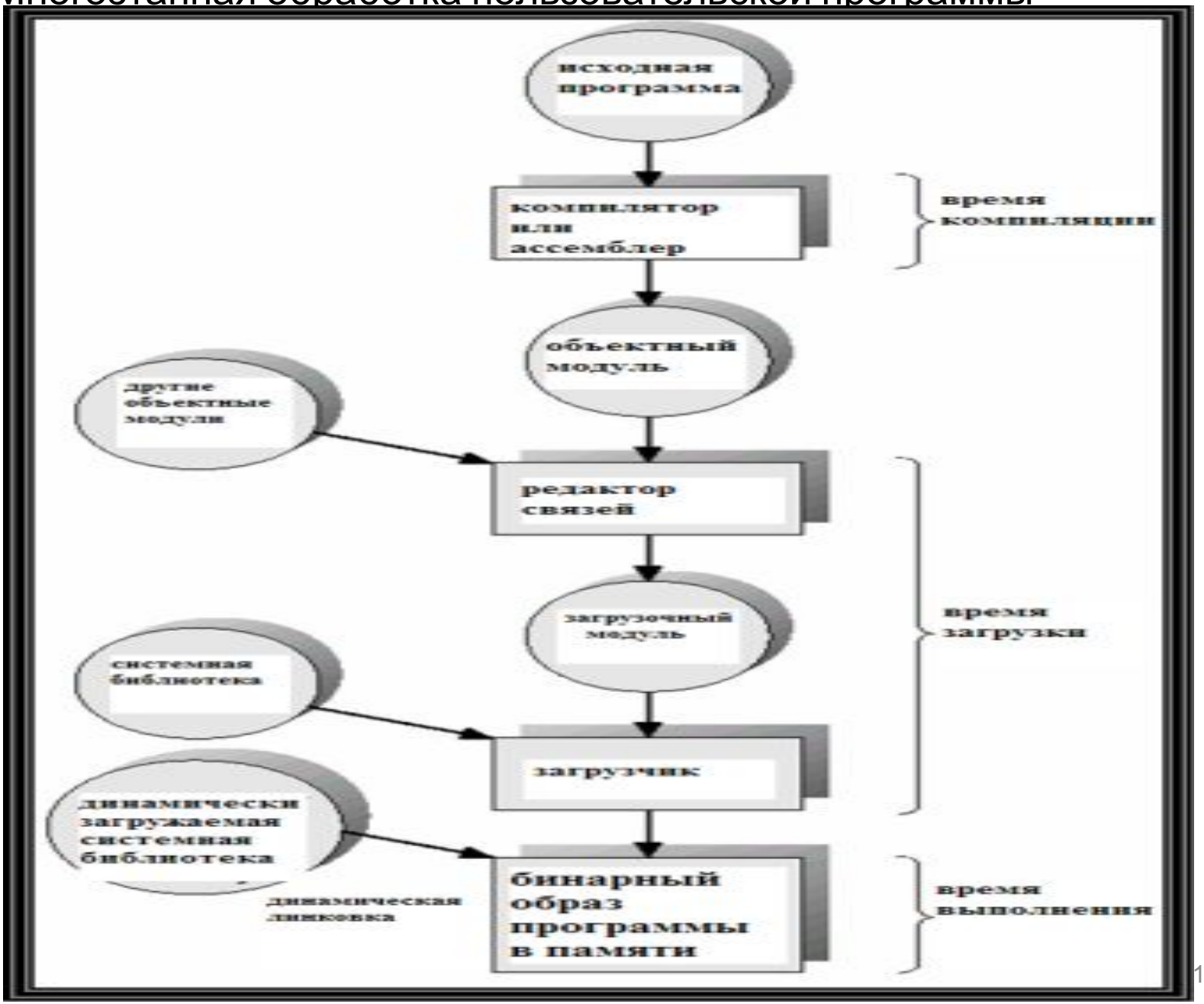
После выбора процесса-"жертвы" с минимальной стоимостью (по одному из приведенных критериев), система прекращает выбранный процесс (процессы), освобождает их ресурсы и выполняет перераспределение ресурсов. Система выполняет "откат" к какому-либо предыдущему безопасному состоянию.

В результате многократного выполнения подобных действий системы, возможно "голодание", так как в качестве жертвы может многократно выбираться один и тот же процесс.

Управление памятью

- **Основные положения размещения процессов в памяти** Любая прога,введ в сист, д.б. размещ в памяти и оформл в виде проца для ее выпо. Каждая прога при вводе в систему помещ во вх очередь – совокупн процессов на диске, ожидающих размещения в памяти для выполнения своих программ. До своего выполнения пользовательские программы проходят в системе несколько стадий.
- **Связывание прог и данных с адресами в памяти** Перед загр данных или кода в память они должны быть в какой-либо момент связ с определ адресами в памяти. Связыв может вып-ся на разных этапах:**-Связ во время компиляции (compile-time).** Если адрес в памяти априорно известен, компилятором может быть сгенерирован код с абсолютными адресами.
- **-Связ во время загрузки (load-time).** Загрузка программы в память – стадия ее обработки системой, предшествующая выполнению программы. **Связывание во время исполнения (runtime),** или **динамическое (позднее) связывание.** Используется, если процесс во время выполнения может быть перемещен из одного сегмента памяти в другой.

Многоэтапная обработка пользовательской программы



- **Исходный код** программы (в форме текстового файла) на языке высокого уровня или на ассемблере преобразуется компилятором или ассемблером в **объектный модуль**, содержащий бинарные выполняемые машинные команды и**таблицу символов**, определенных и использованных в данном модуле кода. Рассмотренная фаза называется**временем компиляции**.
- Следующая фаза обработки программы – **редактирование связей. Редактор связей (linker)** – системная программа, которая получает на вход один или несколько объектных модулей, а на выходе выдает **загрузочный модуль** – двоичный код, образованный кодом нескольких объектных модулей, в котором разрешены все межмодульные ссылки - для каждого символа.
- Загрузочный модуль может быть загружен в память для исполнения с помощью еще одной системной программы – **загрузчика (loader),** который получает на вход загрузочный модуль и файлы с бинарными кодами **системных библиотек,** которые использует программа. Загрузчик, объединяя код программы с кодами системных библиотек, создает **бинарный образ программы в памяти**.

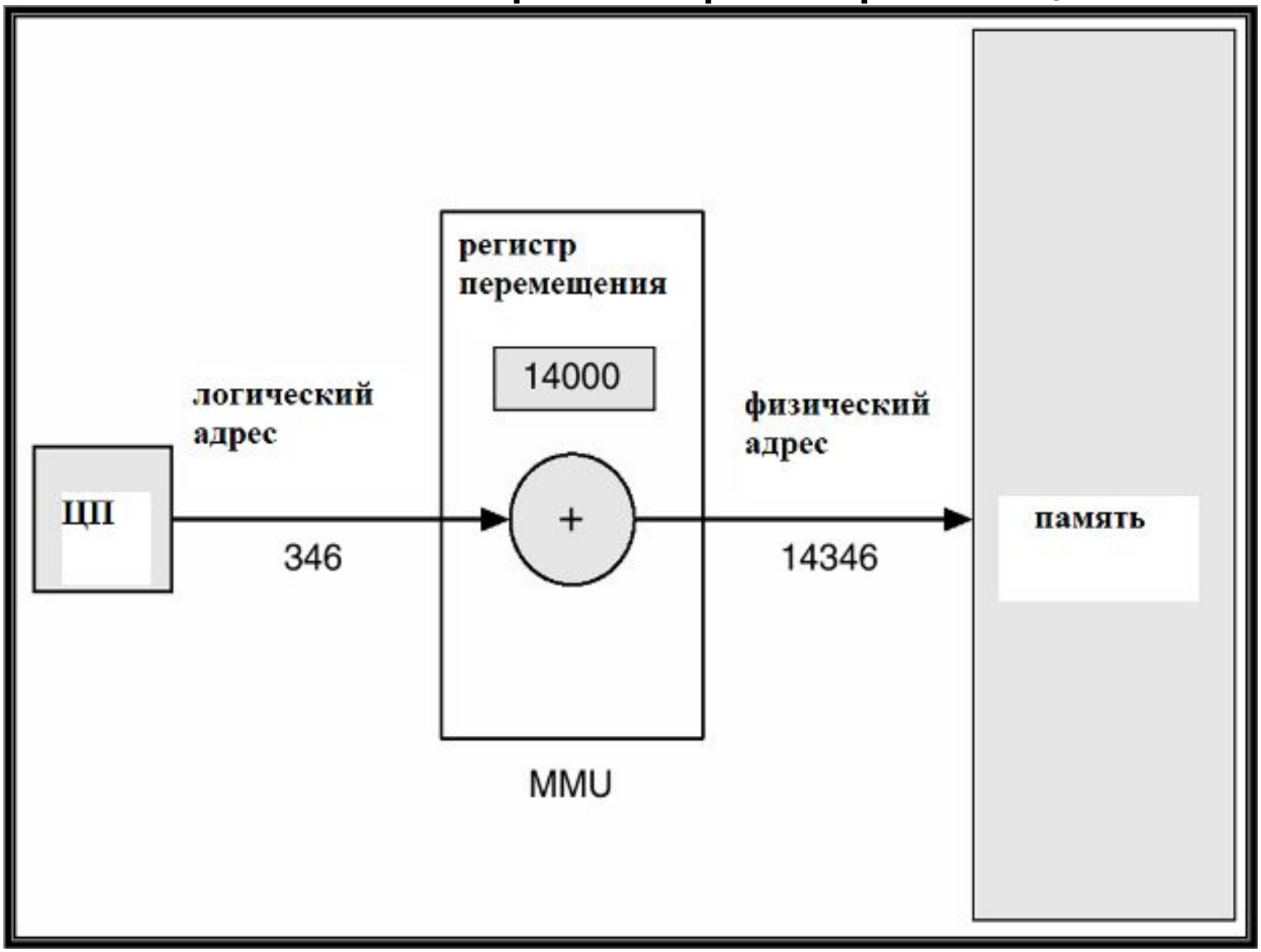
Логическое и физическое адресное пространство

Концепция **логического адресного пространства**, связанного с соответствующим физическим адресным пространством, является одной из основных для управления памятью.

- **Логическим адресом** называется адрес, генерируемый процессором при выполнении машинной команды.
- **Физический адрес –** это реальный адрес в памяти, который "видит" и "понимает" **устройство управления памятью (Memory Management Unit – MMU).**

  Логические адреса совпадают с физическими при связывании адресов во время компиляции или во время загрузки (т.е. **до** исполнения программы). Однако при связывании адресов во время выполнения логические адреса отличаются от физических. Далее рассмотрим этот вопрос подробнее.
- **устройство управления памятью (Memory Management Unit – MMU)** – это один из модулей аппаратуры, отвечающий за адресацию памяти и связанный с процессором и другими устройствами системной шиной.
- Аппаратура MMU использует значение **регистра перемещения**, содержащего адрес начала области памяти, выделенной ОС для программы пользователя.
- Программа пользователя работает только с логическими адресами и не "видит" физических адресов.

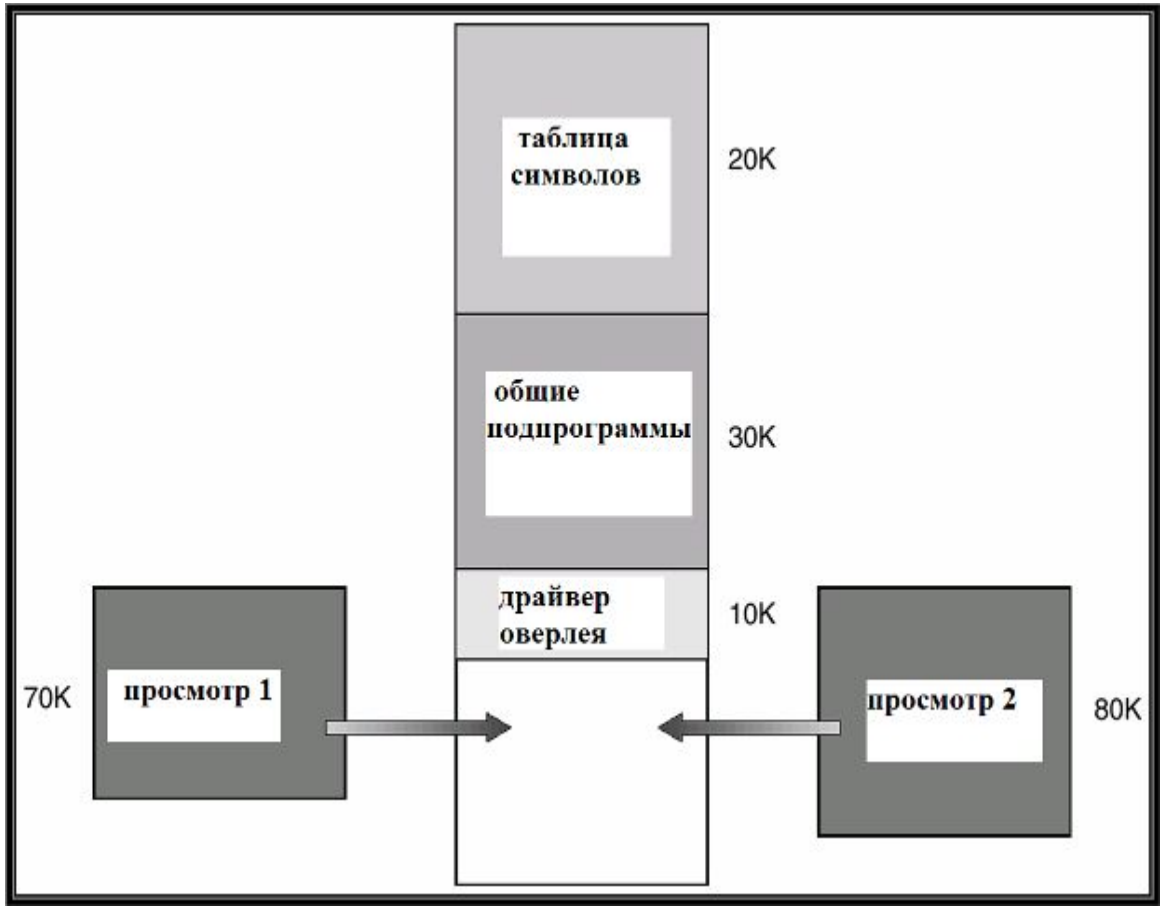
Динамическое перемещение с использованием регистра перемещения



Динамическая загрузка и динамическая линковка

- Под **динамической загрузкой** понимается загрузка подпрограммы в память при первом обращении к ней из пользовательской программы. Это весьма полезный принцип, если требуется сэкономить память, поскольку никакой "лишний" код в этом случае в память не загружается.
- **Линковка** (**linking**) – то же, что и **редактирование связей** и **загрузка**.
- С динамической загрузкой вызываемых подпрограмм тесно связан другой родственный механизм – **динамическая линковка**: линковка во время исполнения программы. В коде программы размещается **заглушка для исполнения (execution stub) –** небольшой фрагмент кода, выполняющий системный вызов модуля ОС, размещающего в памяти код динамически линкуемой библиотечной подпрограммы.

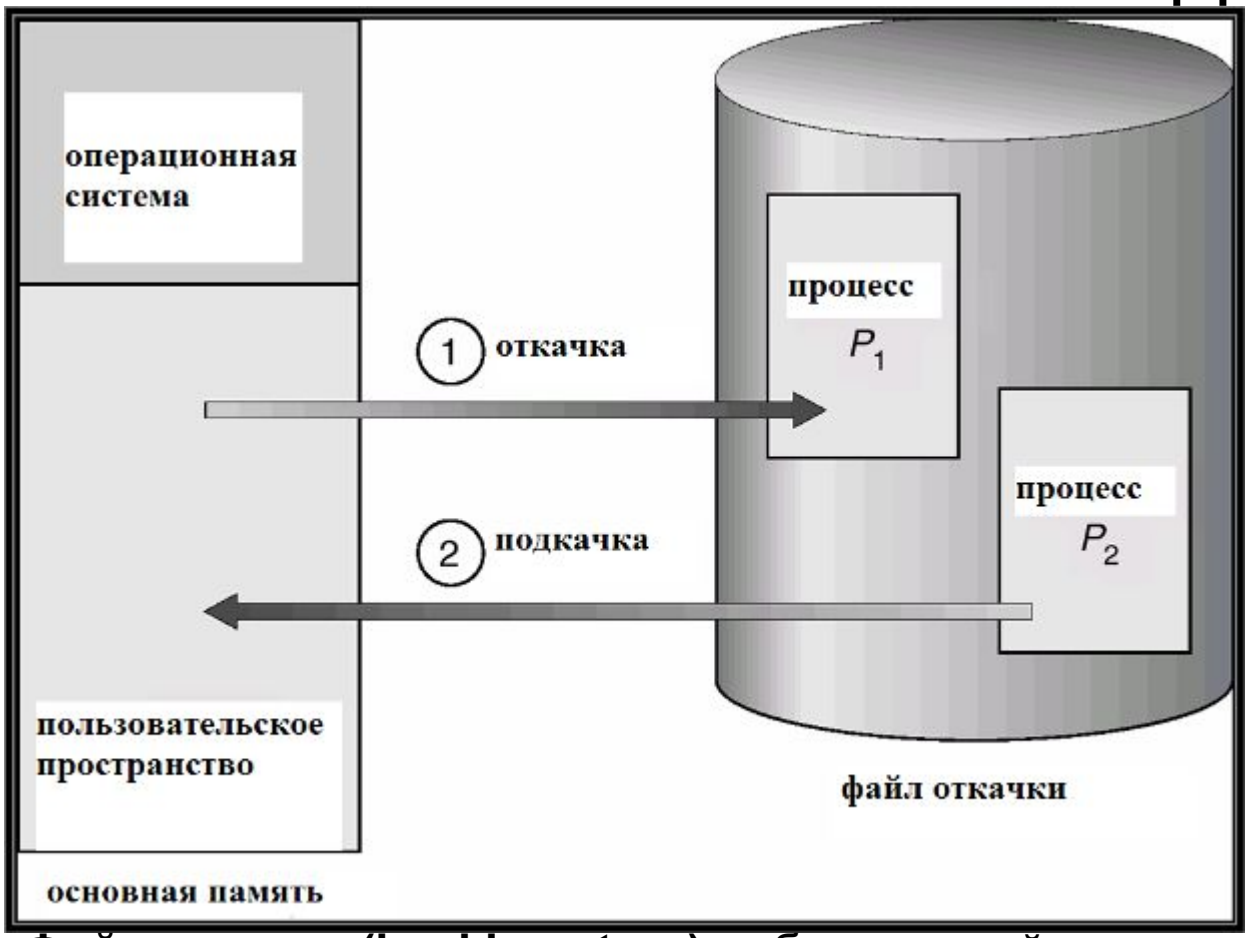
Оверлейная структура для двухпросмотрового ассемблера



специальный системный механизм, называемый **оверлейная структура (overlay)** - организация программы при недостаточном объеме основной памяти, при которой система выполняет поочередную загрузку в одну и ту же область памяти то одной, то другой исполняемой группы модулей программы.

Типичный для ранних компьютеров и ОС пример программы с оверлейной структурой – двухпросмотровый ассемблер.

Схема откачки и подкачки

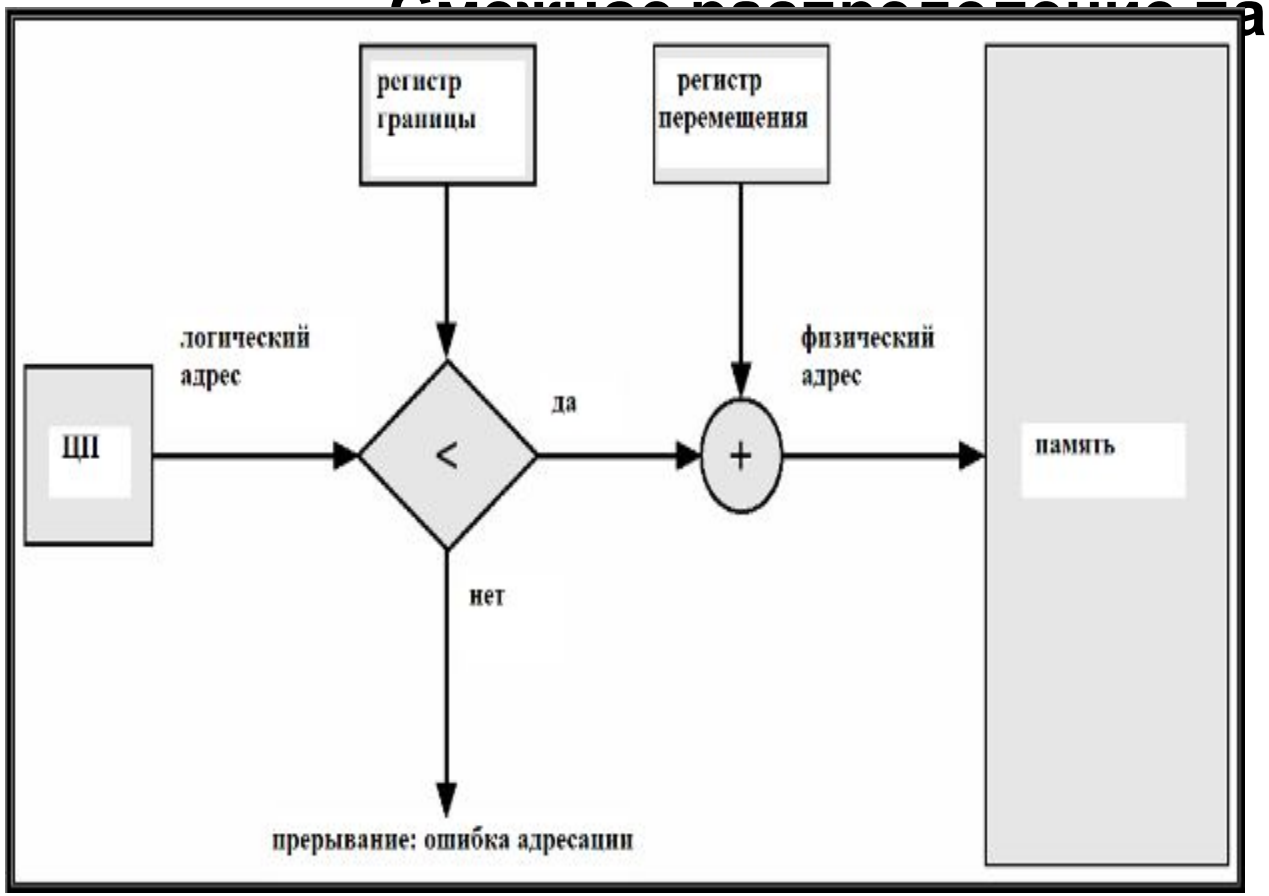


Откачка и подкачка (swapping) – это действия операционной системы по **откачке (записи)** образа неактивного процесса на диск или **подкачке (считыванию)** активного процесса в осн память. Необхть вып подобных действий вызвана нехваткой осн памяти.

Файл откачки (backing store) - обл дисковой памяти, используемая операционной системой для хранения образов откачанных процессов. Файл откачки организ макс эффективно: обеспечивается прямой доступ ко всем образам процессов в памяти (например, через таблицу по номеру процесса).

Аппаратная поддержка регистров перемещения и границы

Смежное распределение памяти



Наиболее простая и распространенная стратегия распределения памяти – **смежное распределение памяти** – распределение памяти для пользовательских процессов в одной смежной области памяти.

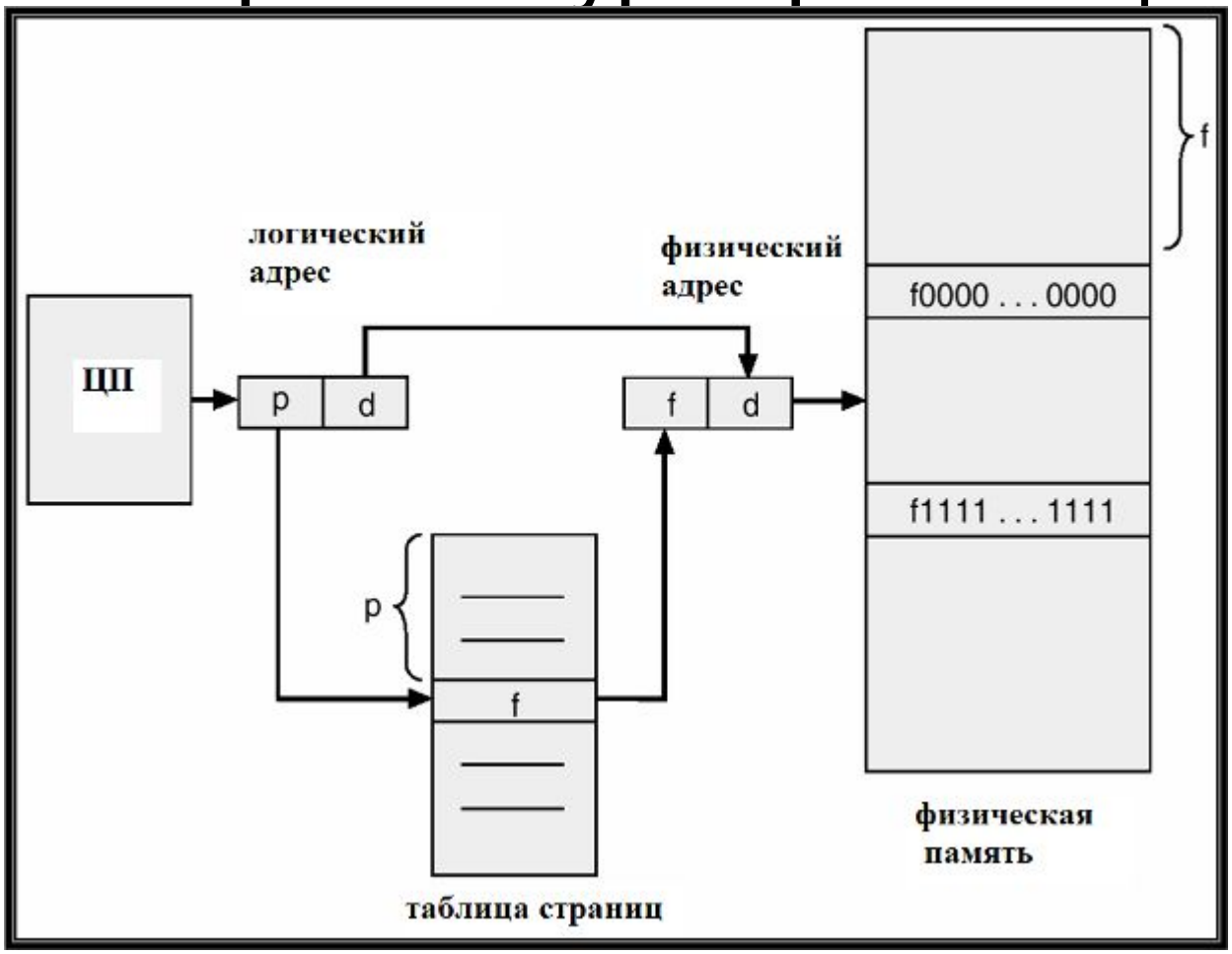
Основная память разбивается на две смежных части (partitions), которые "растут" навстречу друг другу: резидентная часть ОС и вектор прерываний – по меньшим адресам, пользовательские процессы – по адресам.

Фрагментация

Фрагментация – это дробление памяти на мелкие не смежные свободные области маленького размера. Фрагментация возникает после выполнения системой большого числа запросов на память, таких, что размеры подходящих свободных участков памяти оказываются немного больше, чем требуемые.

- Фрагментация бывает **внутренняя** и **внешняя**.
- **внешней фрагментации** имеется достаточно большая область свободной памяти, но она не является непрерывной.
- **Внутренняя фрагментация** может возникнуть вследствие применения системой специфической стратегии выделения памяти, при которой фактически в ответ на запрос память выделяется несколько большего размера, чем требуется, - например, с точностью до **страницы (листа**), размер которого – степень двойки. Страничная организация памяти подробно рассматривается далее в данной лекции.
- Внешняя фрагментация может быть уменьшена или ликвидирована путем применения **компактировки (compaction) –**сдвига или перемешивания памяти с целью объединения всех не смежных свободных областей в один непрерывный блок.
- При компактировке памяти и анализе свободных областей может быть выявлена **проблема зависшей задачи**: какая-либо задача может "застрять" в памяти, так как выполняет ввод-вывод в свою область памяти (по этой причине откачать ее невозможно). Решение данной проблемы: **ввод-вывод должен выполняться только в специальные буфера**,выделяемой для этой цели операционной системой.

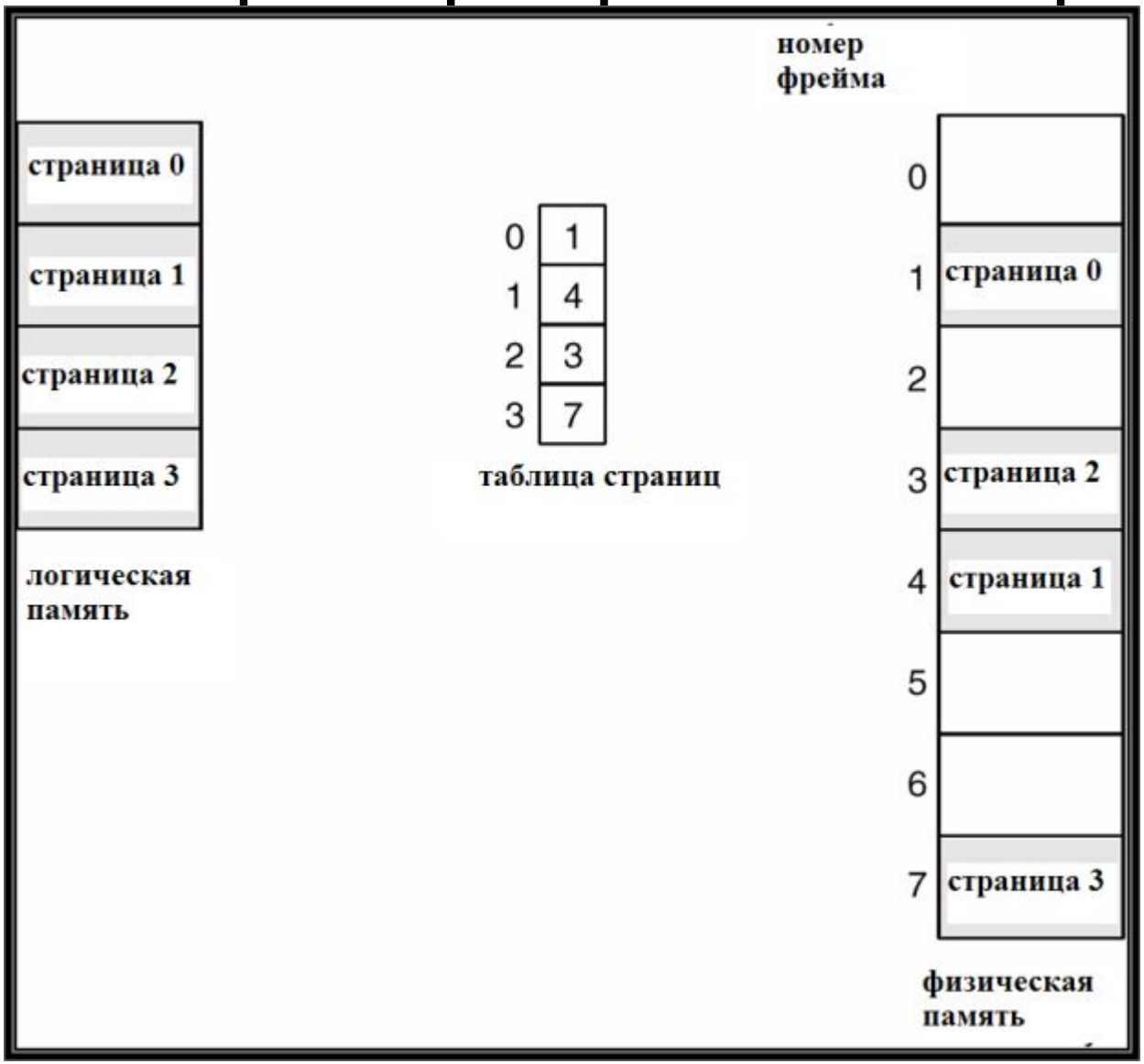
Архитектура трансляции адресов



адрес обрабатывается системой как структура **(p, d)**:его старшие разряды обозначают **номер страницы**, младшие – **смещение внутри страницы. Номер страницы (p)** как индекс в таблице страниц, соотв элемент кот содержит базовый **адрес начала страницы в физической памяти**.

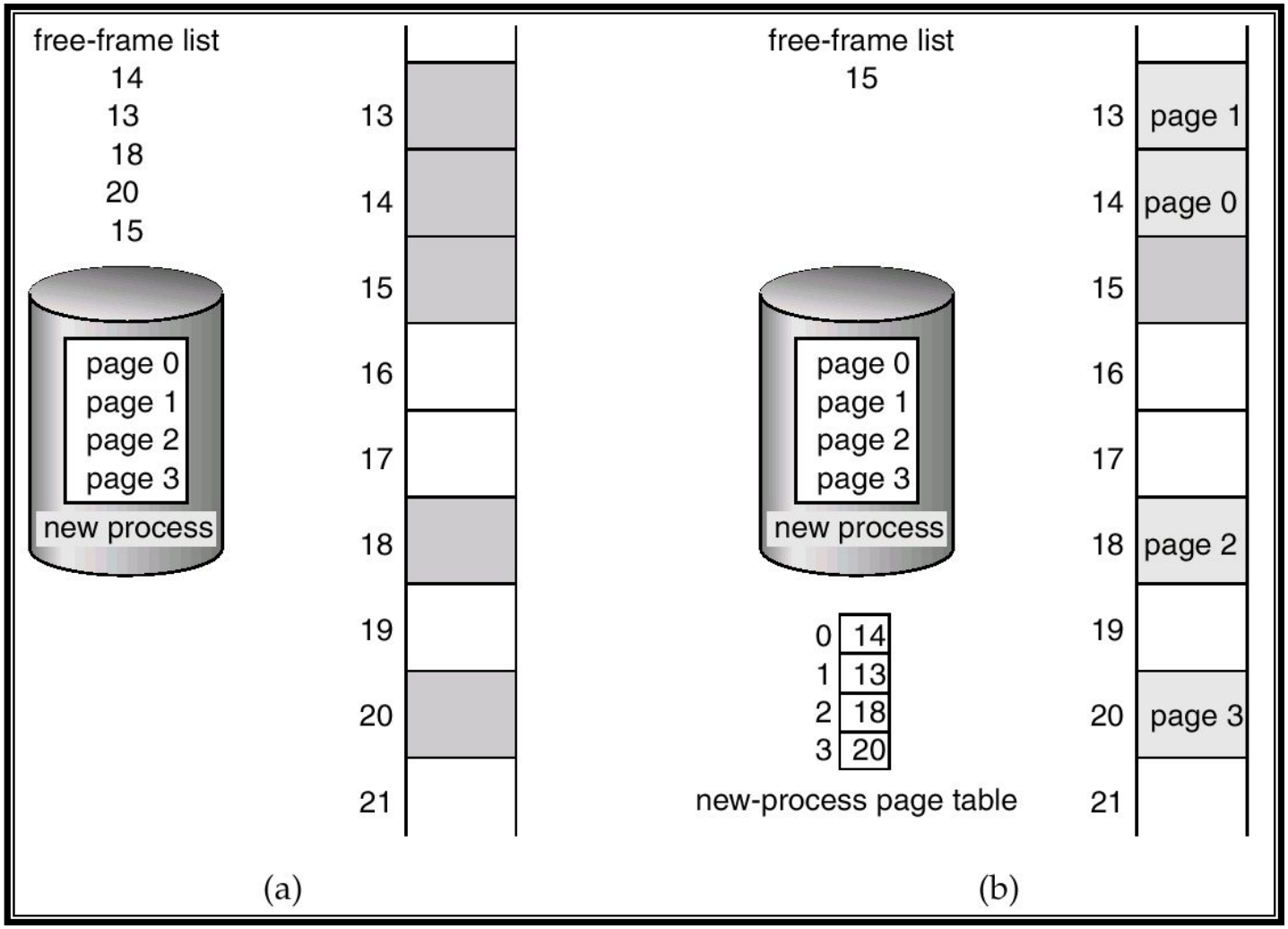
Смещение внутри страницы (d) добавляется к ее базовому адресу. В результате формируется физический адрес, передаваемый в устройство управления памятью.

Пример страничной организации

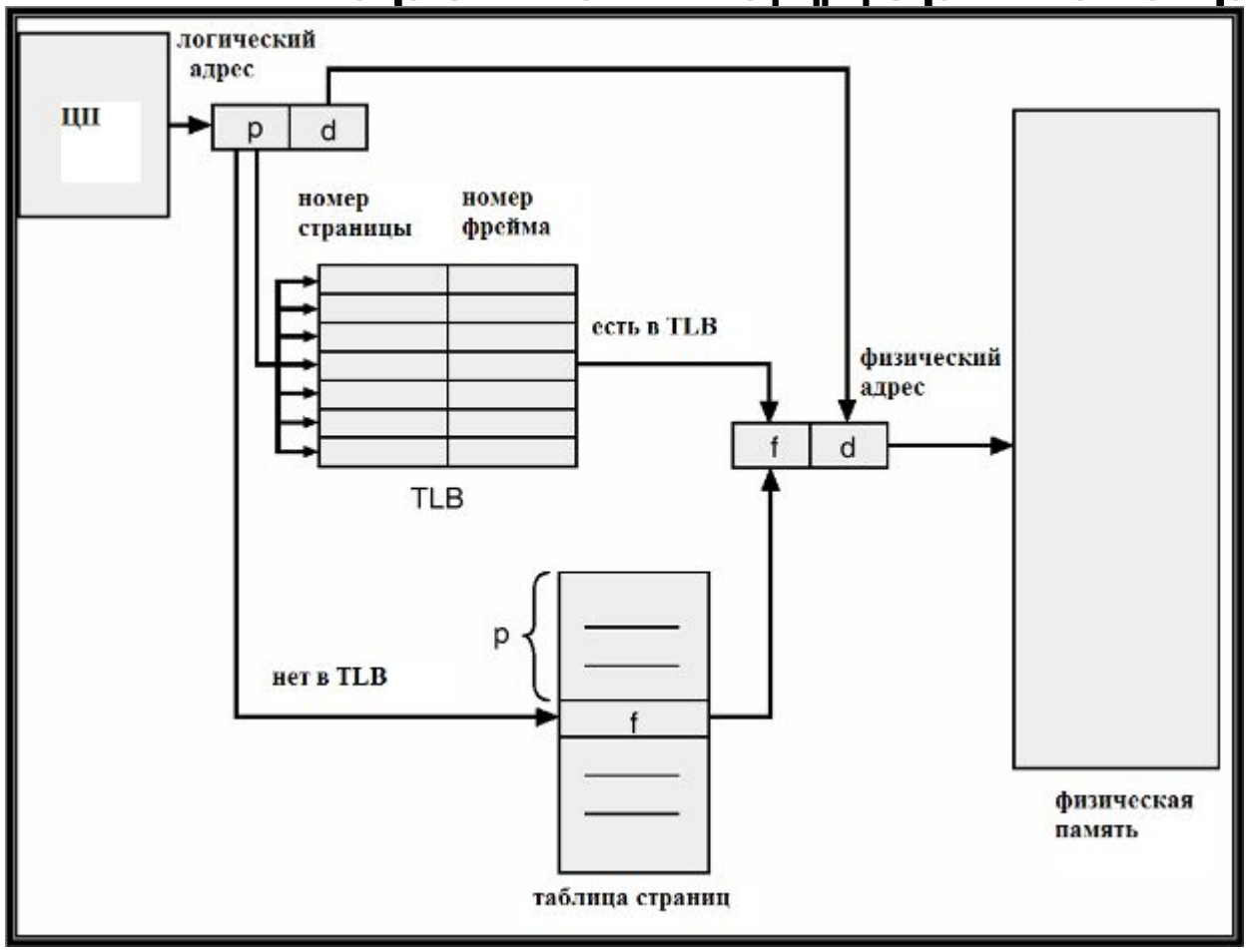


в отличие от непрерывной логической памяти процесса, соотв фреймы стр в осн памяти м.б. расп не смежно: логичстранице 0 соотв фрейм 1, странице 1 – фрейм 4, странице 2 – фрейм 3, странице 3 – фрейм 7.

Список свободных фреймов



Аппаратная поддержка страничной



Использование ассоциативной памяти. Таблица страниц – непрерыв область физической памяти. В системе имеется **базовый регистр таблицы страниц (page table base register – PTBR)**,указывающий на таблицу страниц и хранящий ее длину. *Таким образом, при страничной организации любой доступ к памяти требует фактически не одного, а двух обращений в память – одно в таблицу страниц, другое – непосредственно к данным или команде. (недостаток)*

Пробла 2 обращ решается введением **ассоц памяти (cache) страниц**,называемой также **буфер трансляции адресов (translation lookaside buffer – TLB**.Ассоц пам явл ассоци списком пар вида: **(номер страницы, номер фрейма)**.Ее быстродействие значительно выше, чем у основной памяти и у регистров.

Эффективное время поиска (effective access time – EAT)

- **Ассоциативный поиск = ε единиц времени**
- **Предположим, что цикл памяти – 1 микросекунда**
- **Hit ratio – отношение, указывающее, сколько раз (в среднем) номер страницы будет найден по ассоциативным регистрам. Отношение зависит от размера кэш-памяти .**
- **Hit ratio = α**
- TLB- эмпирическую вероятность нахождения номера страницы в ассоциативной памяти.
- Вычислим математическое ожидание времени доступа – **Effective Access Time (EAT)**.Вероятность того, что номер страницы **не** будет найден в TLB, равна 1 – . Тогда получим:
- **Effective Access Time (EAT)**

  EAT = (1 + ε) α + (2 + ε)(1 – α)

  = 2 + ε – α

Бит valid/invalid в таблице страниц

В примере процесс имеет 6 логич стран с номерами от 0 до 5.

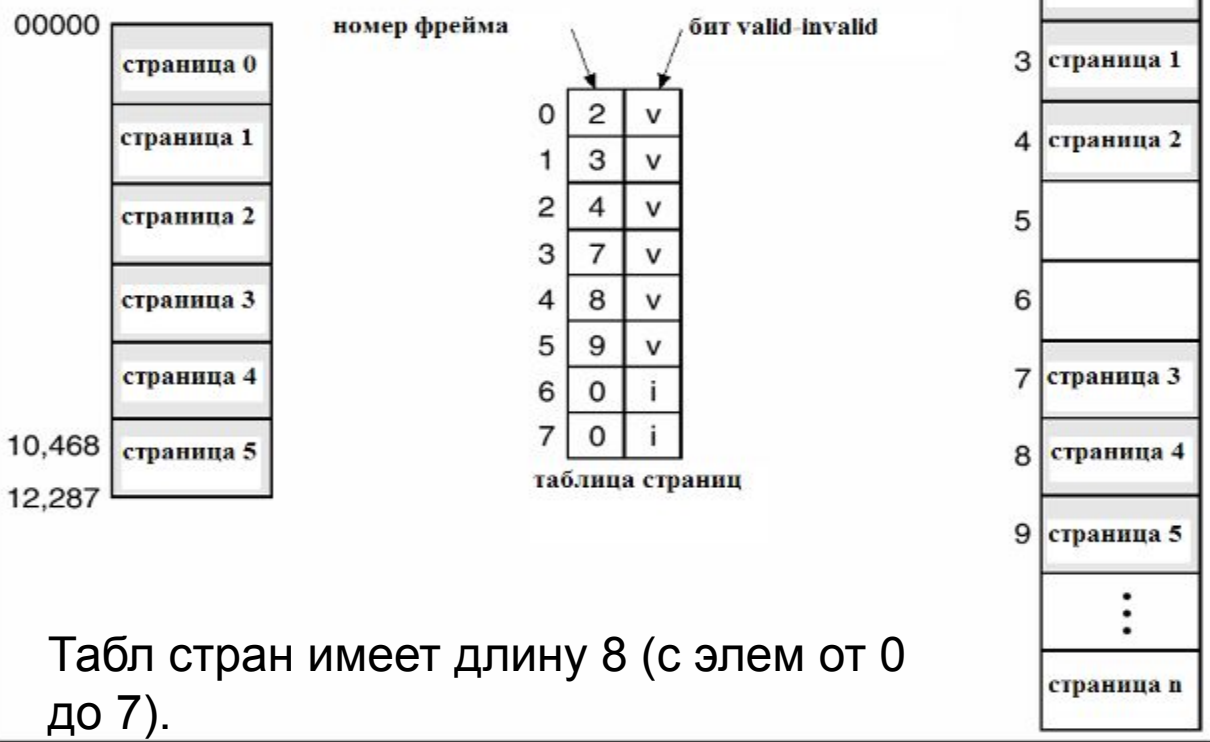


Табл стран имеет длину 8 (с элем от 0 до 7).

Защита памяти

При адресации с пом стран организ возможно, что логич адрес сформирован неверно, и его номер стран вых за пределы лог пам проца. Защита от неверной адресации может быть реализ хранением и проверкой дополнит **бита valid-invalid** в каждом элементе таблицы страниц. Значение **valid** указ, что стран с данным номером принадлеж логич пам проца, значение **invalid** – что это не так.

Элты 6 и 7 не соотв логич стран проца, поэтому в них биты valid-invalid устан взнач **invalid**. Поэтому при попытке обращения по логич адресу с номером стран 6 или 7 произойдет прерывание по неверной адресации.

Схема двухуровневых таблиц страниц

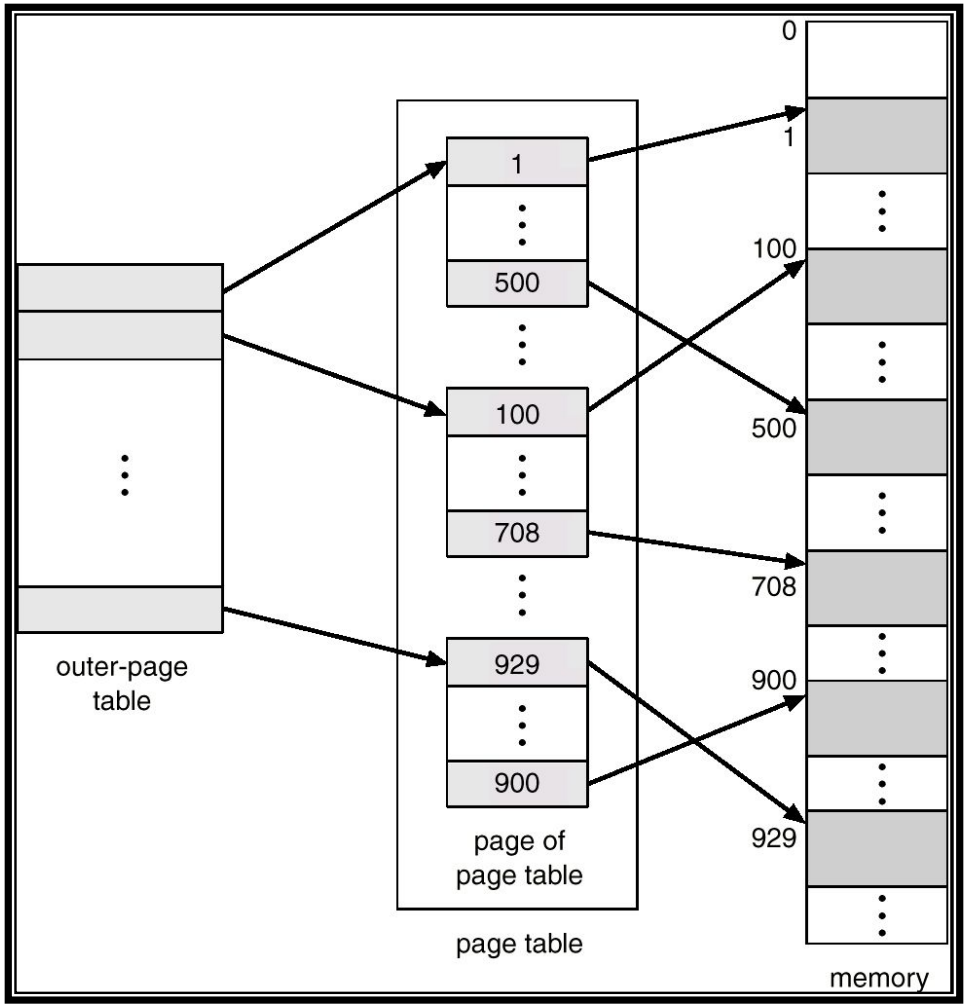
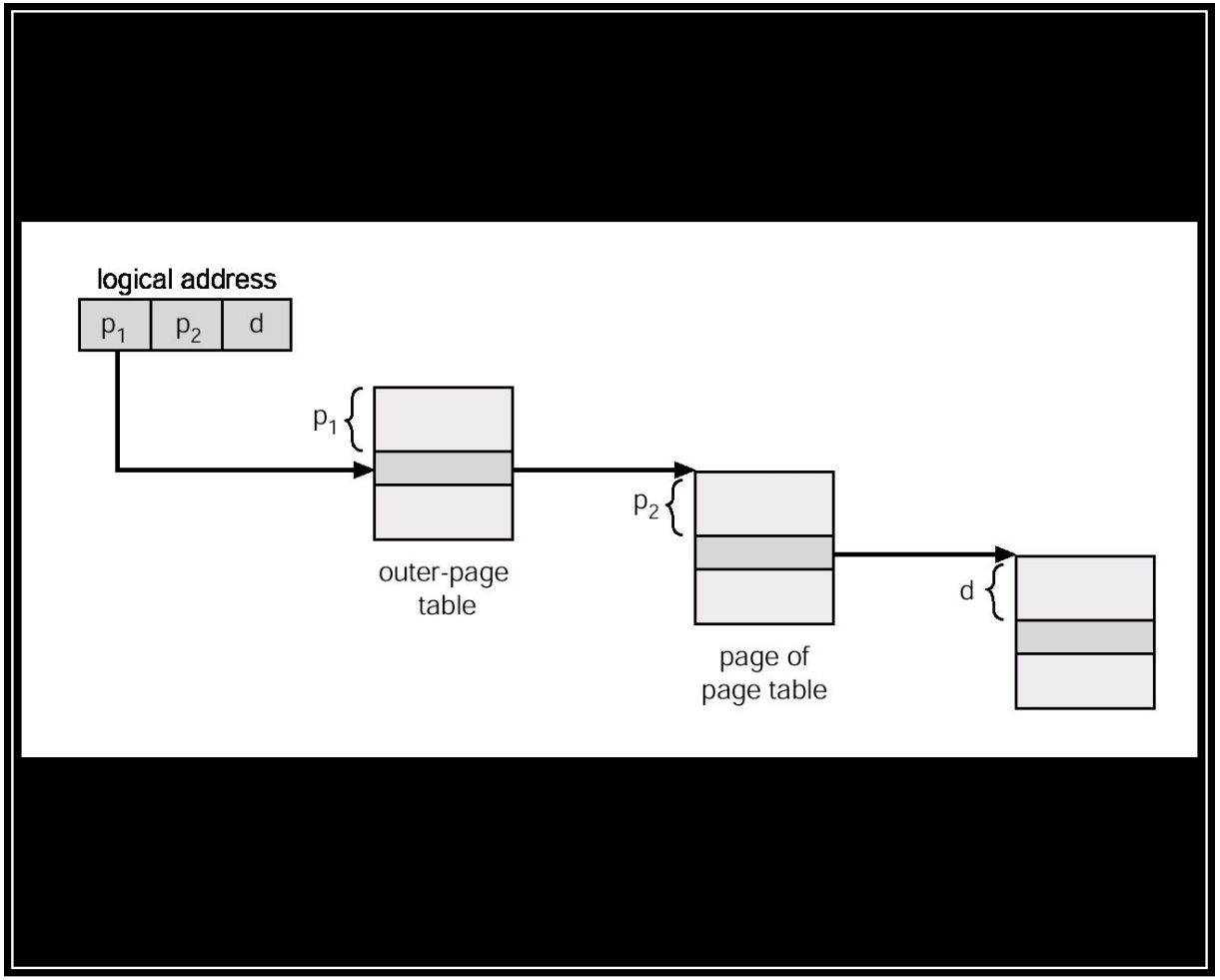
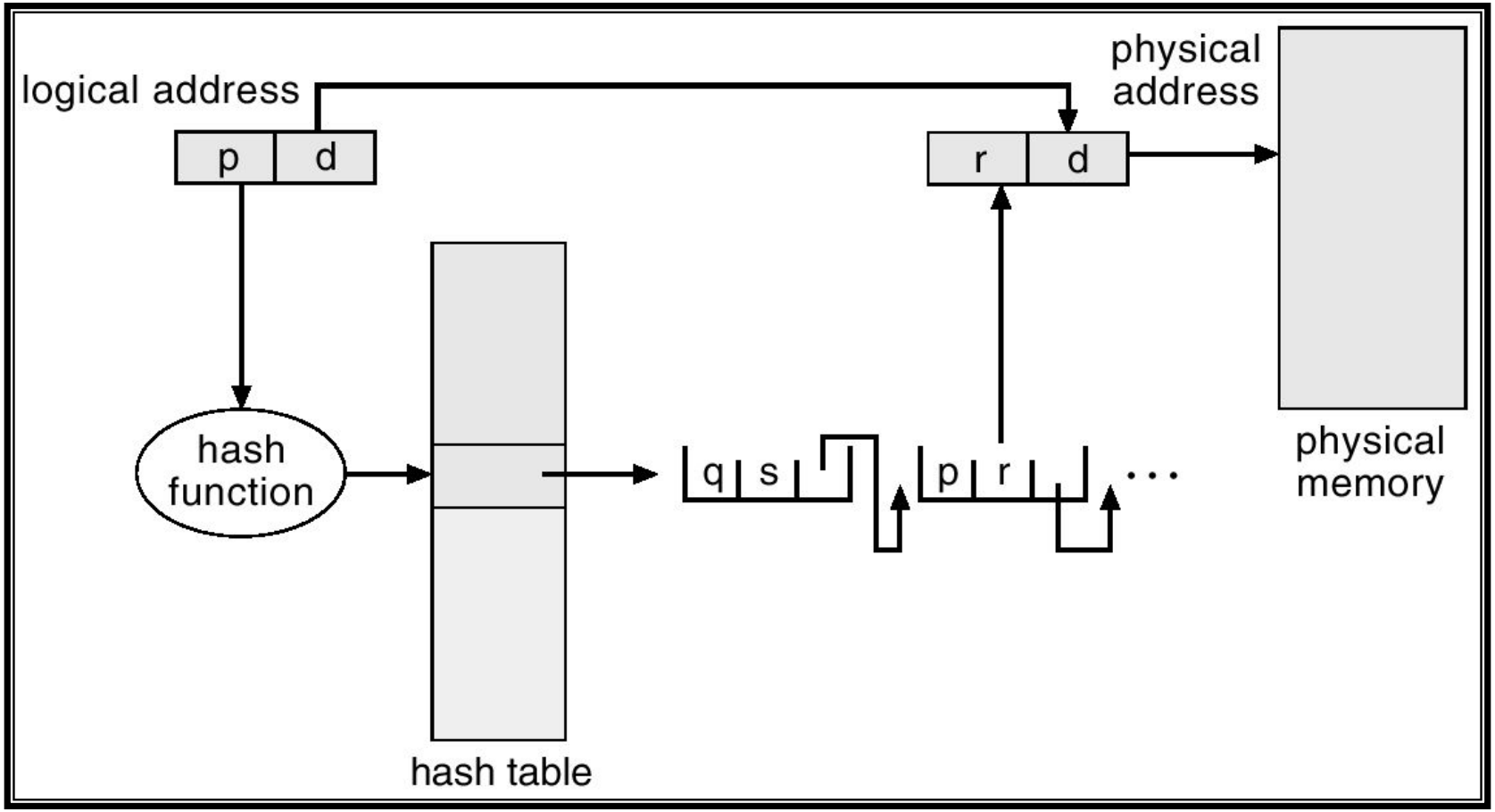


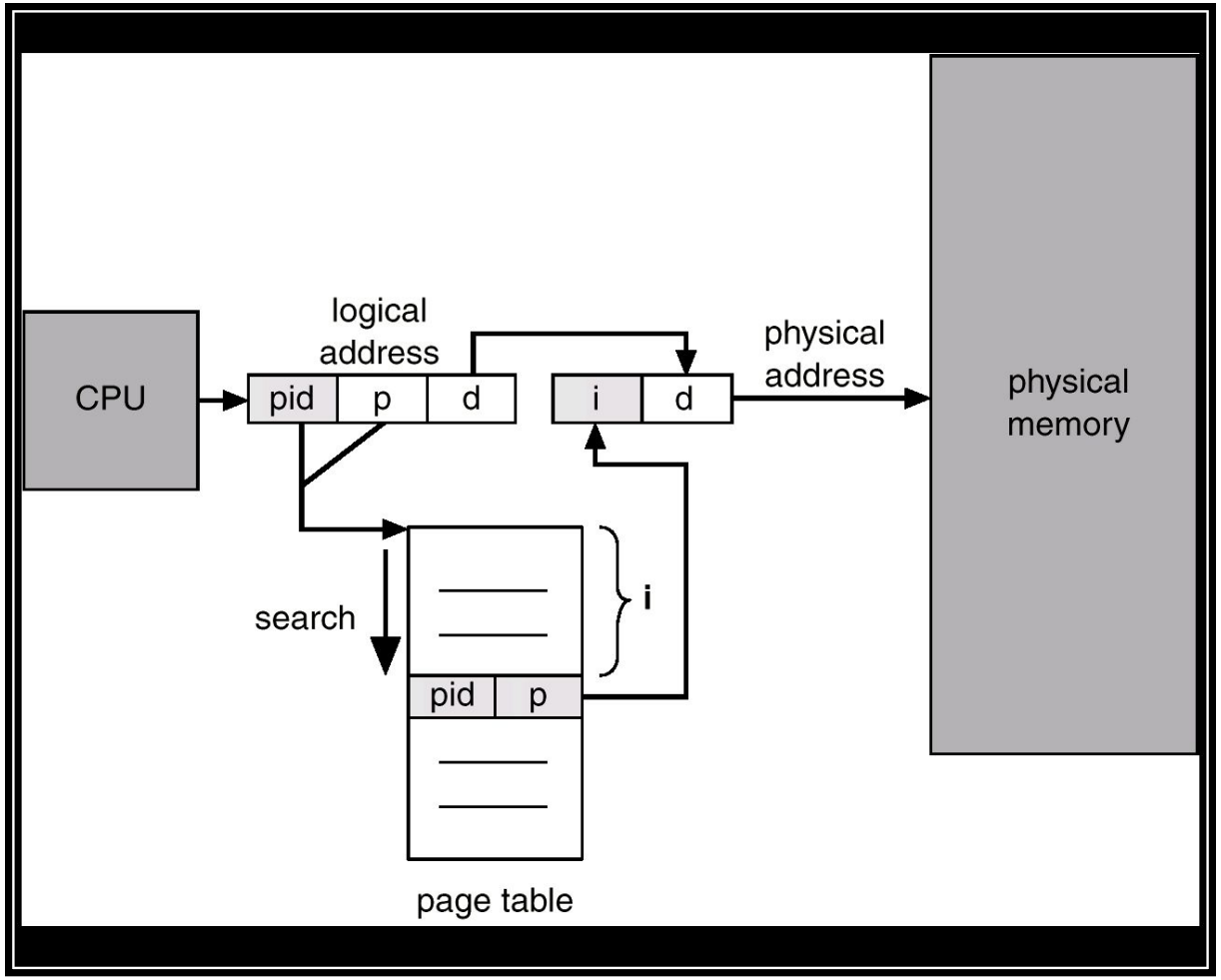
Схема адресной трансляции



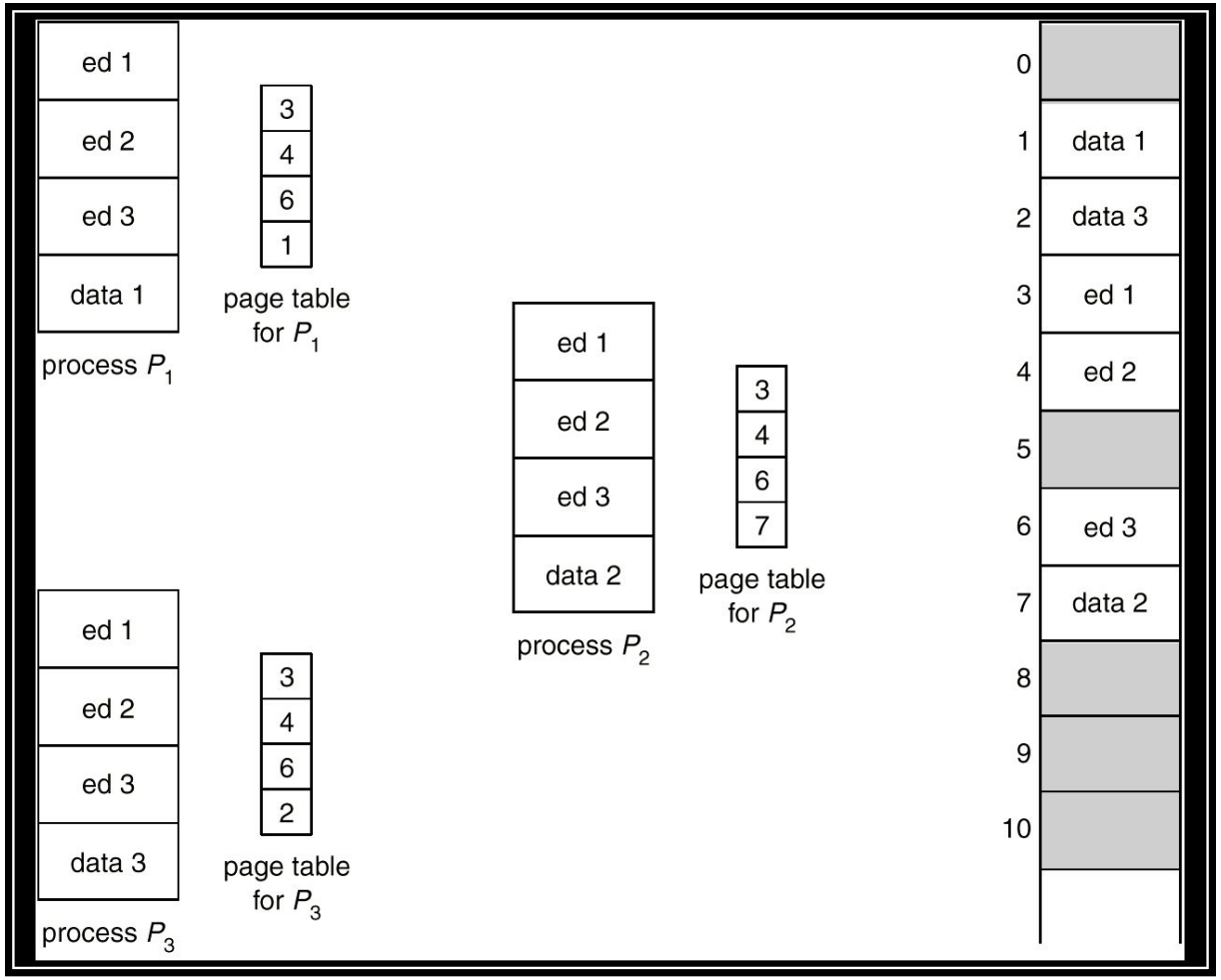
Хешированная таблица страниц



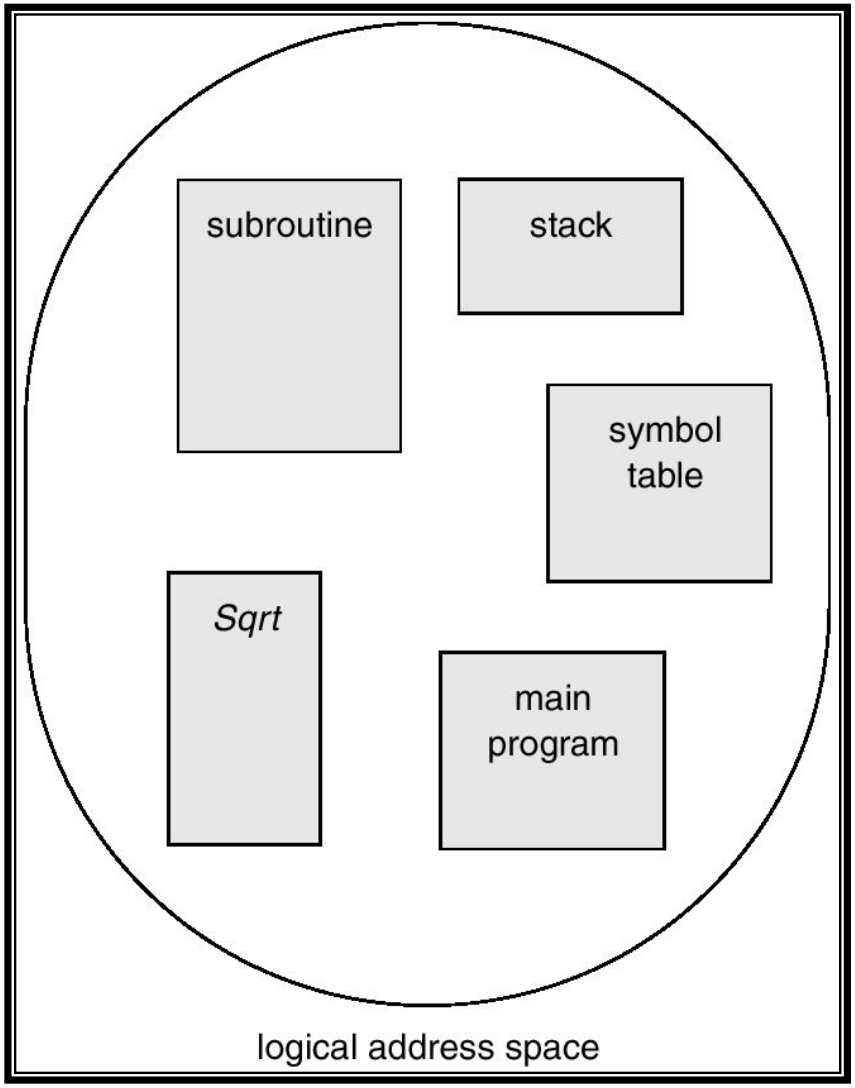
Архитектура инвертированной таблицы страниц



Пример: разделяемые страницы



Программа с точки зрения пользователя



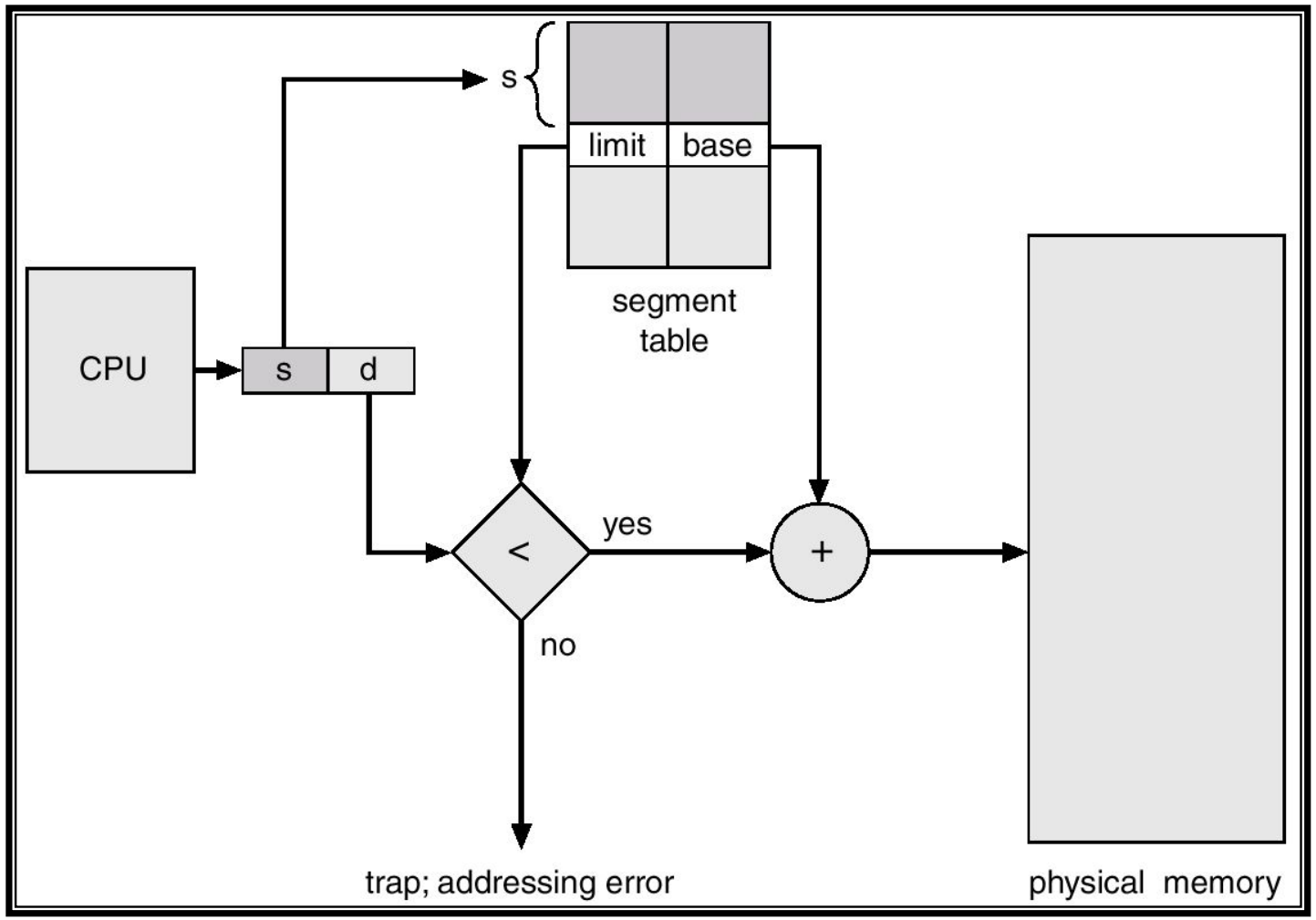
Архитектура сегментной организации памяти

- **Логический адрес ~ пара:**

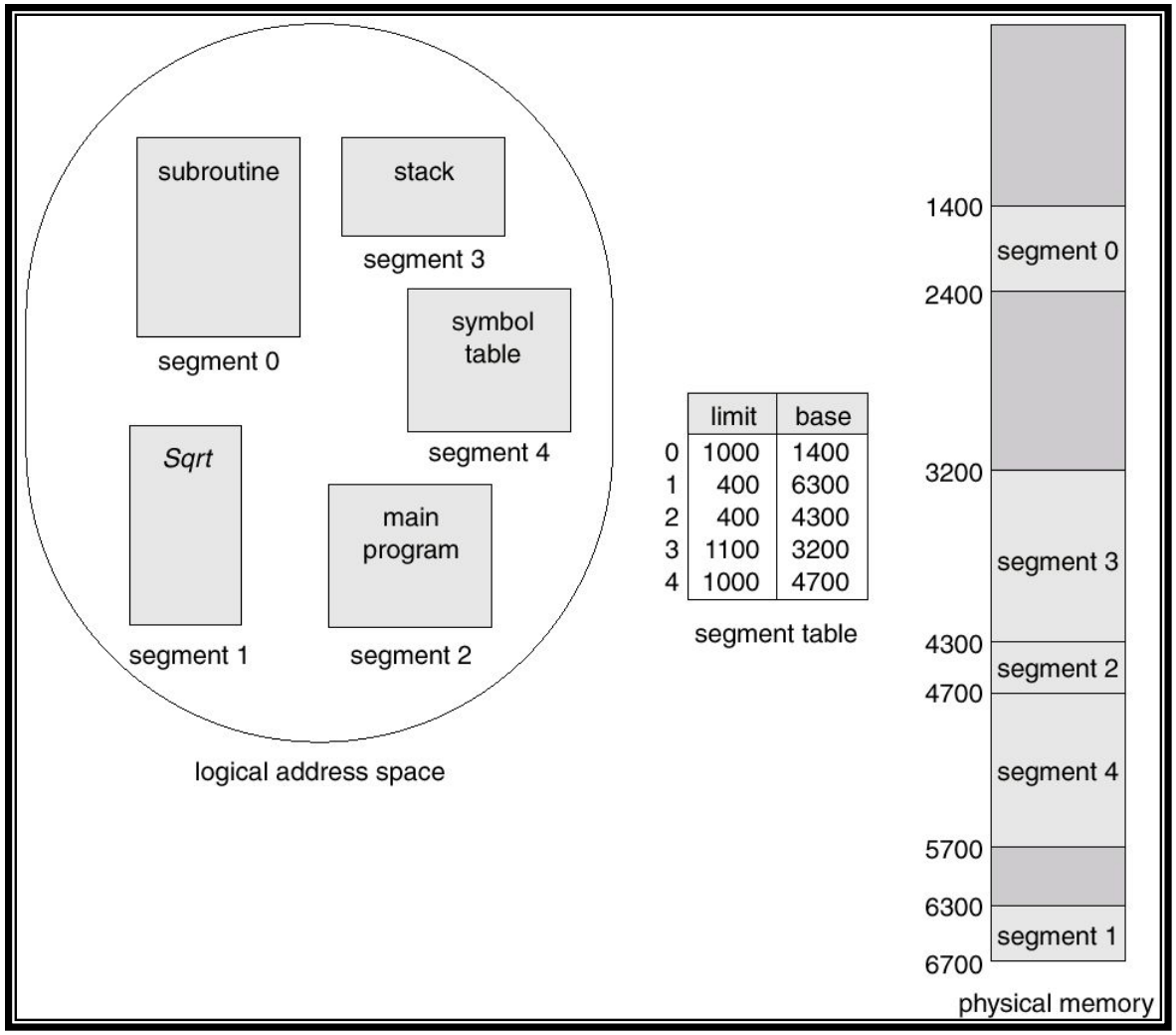
  <segment-number, offset>,
- ***Таблица сегментов* – служит для отображения логических адресов в физические; каждый ее элемент содержит:**
  - ***base* – начальный адрес сегмента в оперативной (физической) памяти.**
  - ***limit* – длину сегмента.**
- ***Базовый регистр таблицы сегментов* - *segment-table base register (STBR)* срдержит адрес таблицы сегментов в памяти.**
- ***Регистр длины таблицы сегментов* - *segment-table length register (STLR)* содержит число сегментов, используемое программой;**

  номер сегмента *s* корректен, если *s* < STLR.

Аппаратная поддержка сегментного распределения памяти



Пример сегментной организации памяти



Совместное использование сегментов

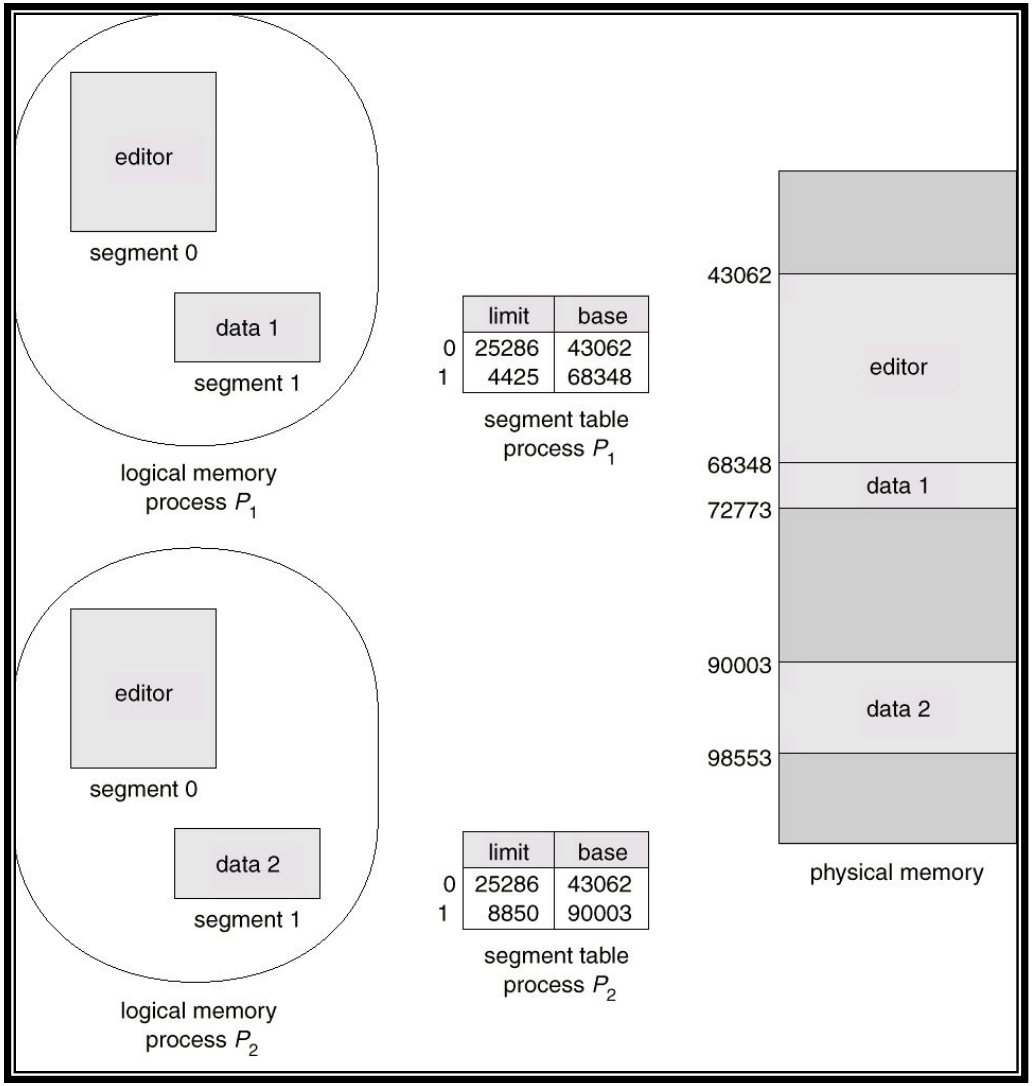
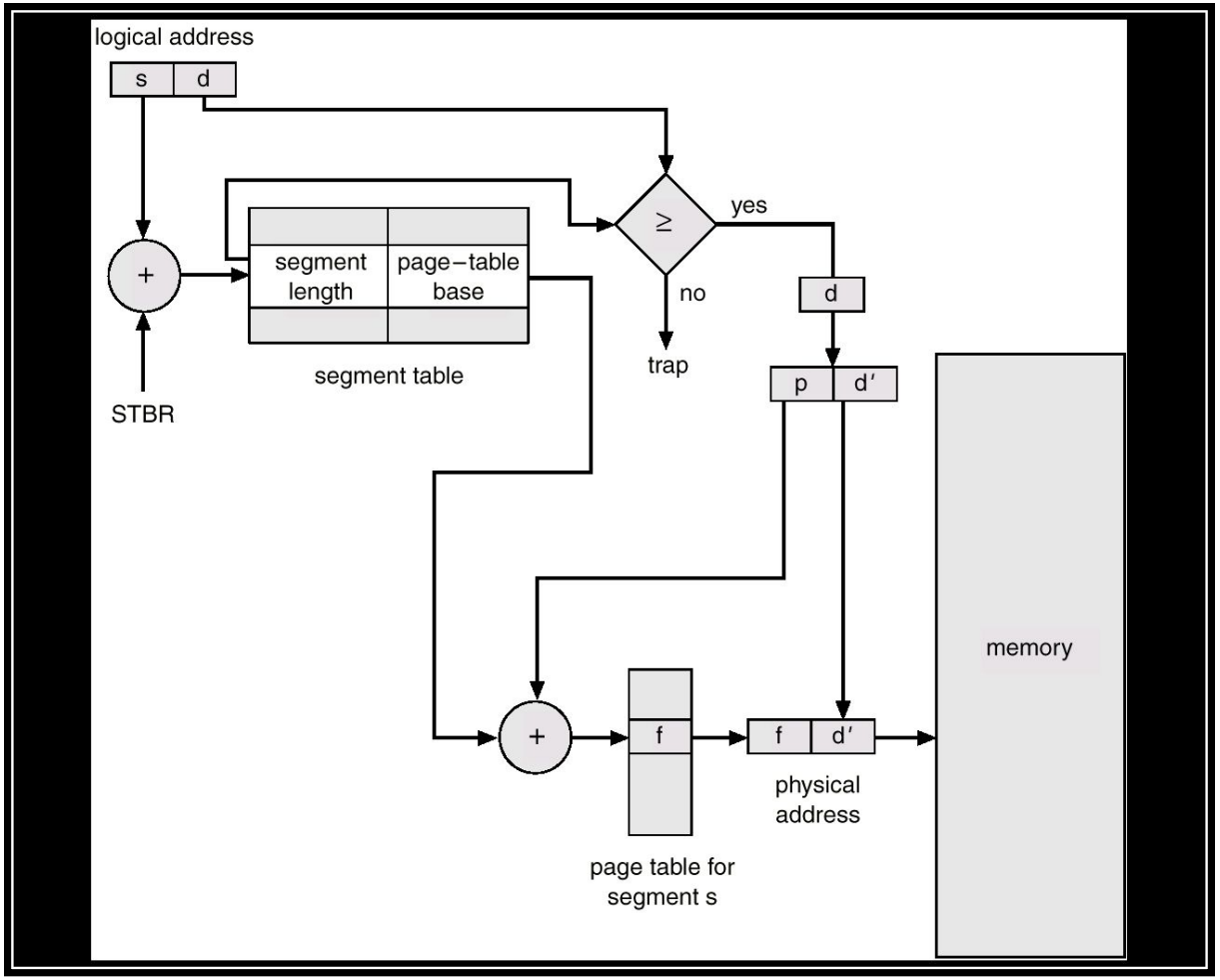
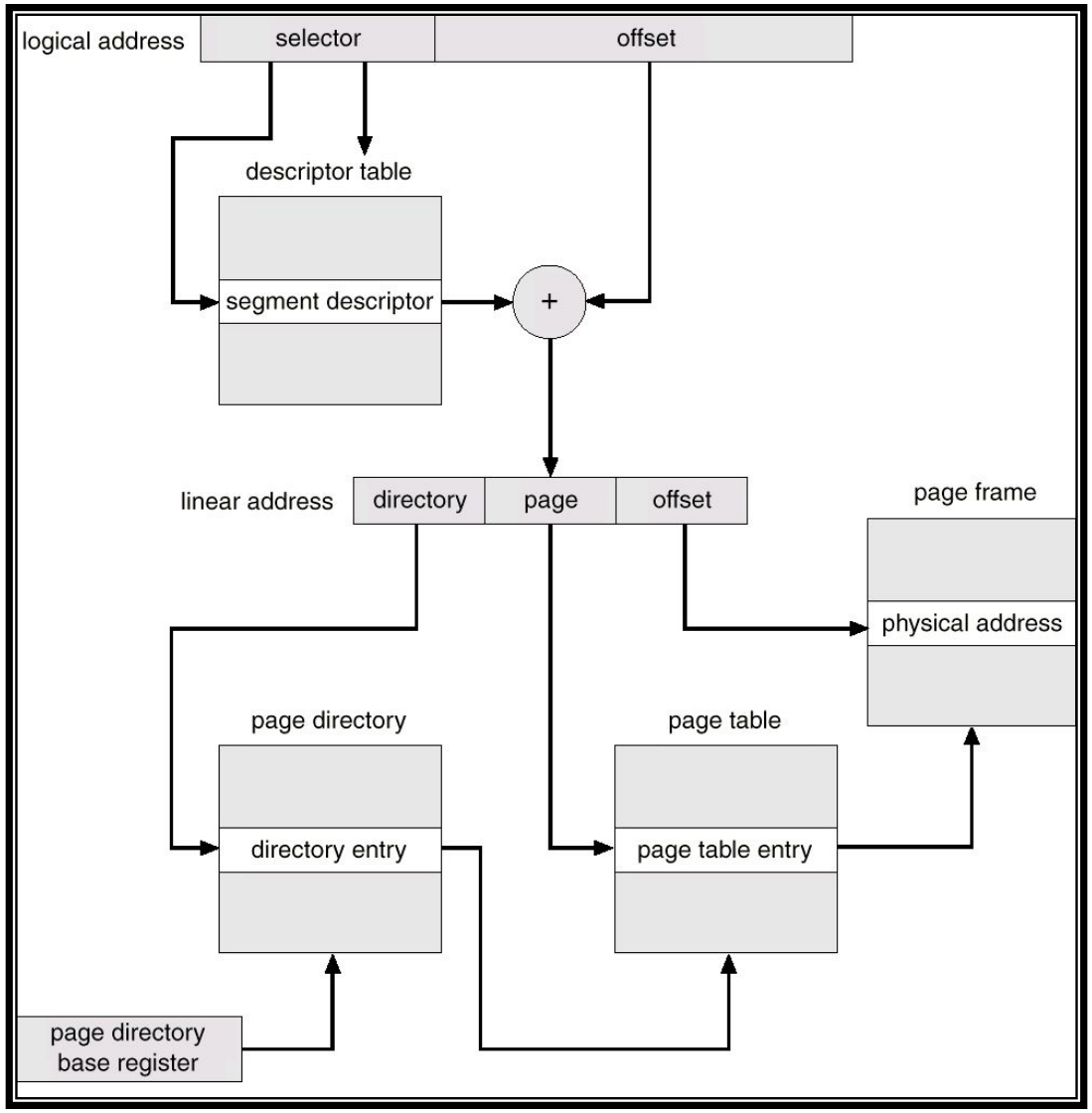


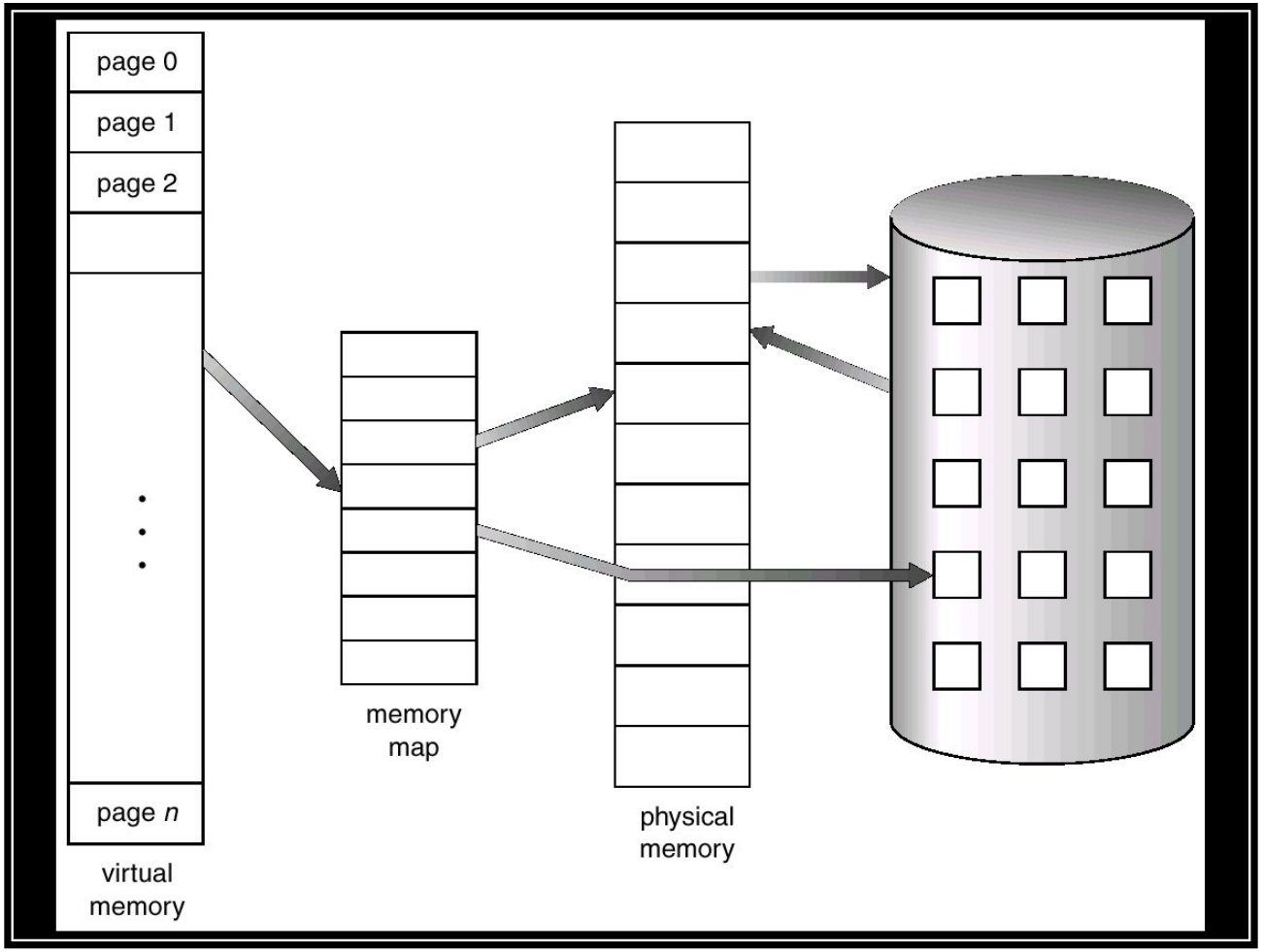
Схема трансляции адресов в MULTICS



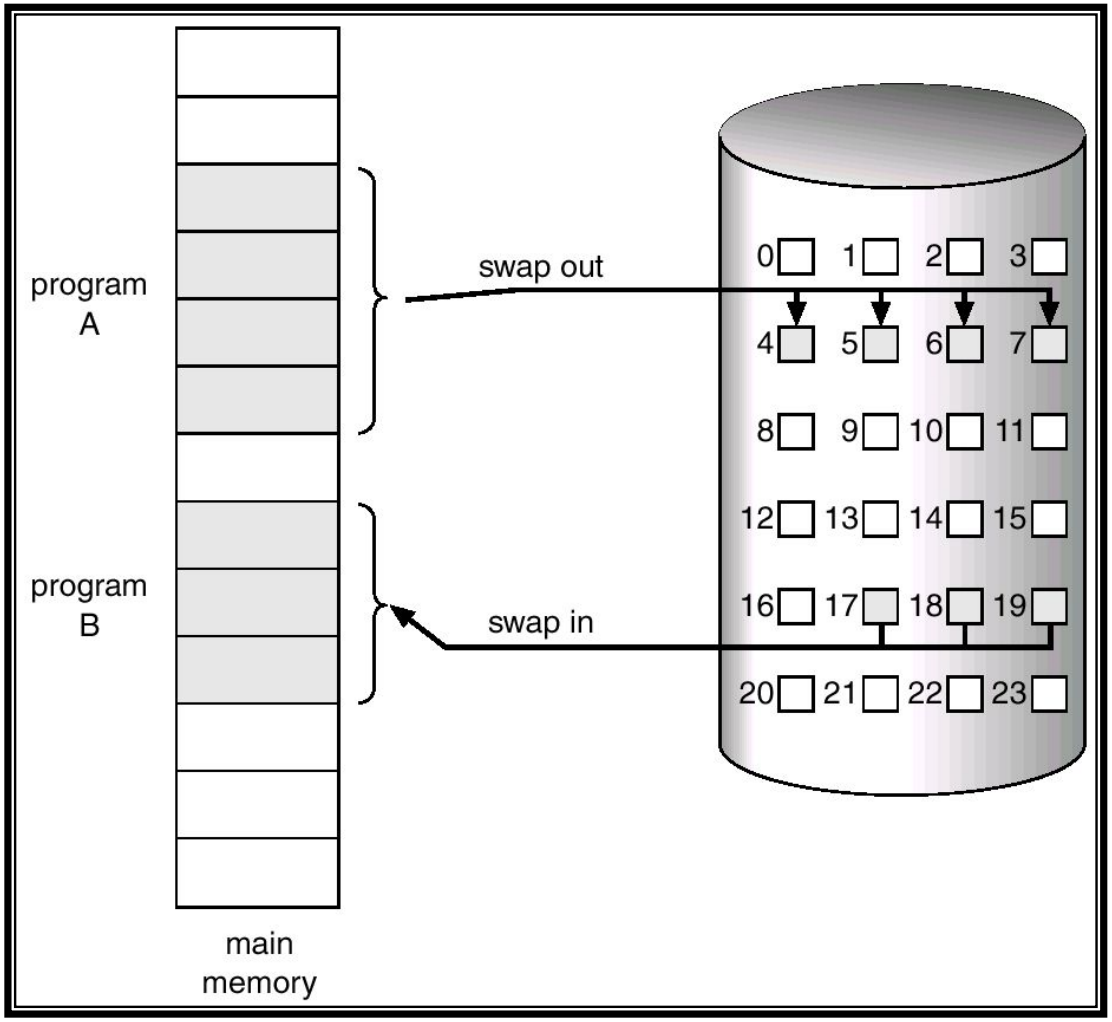
Трансляция адресов в Intel 386



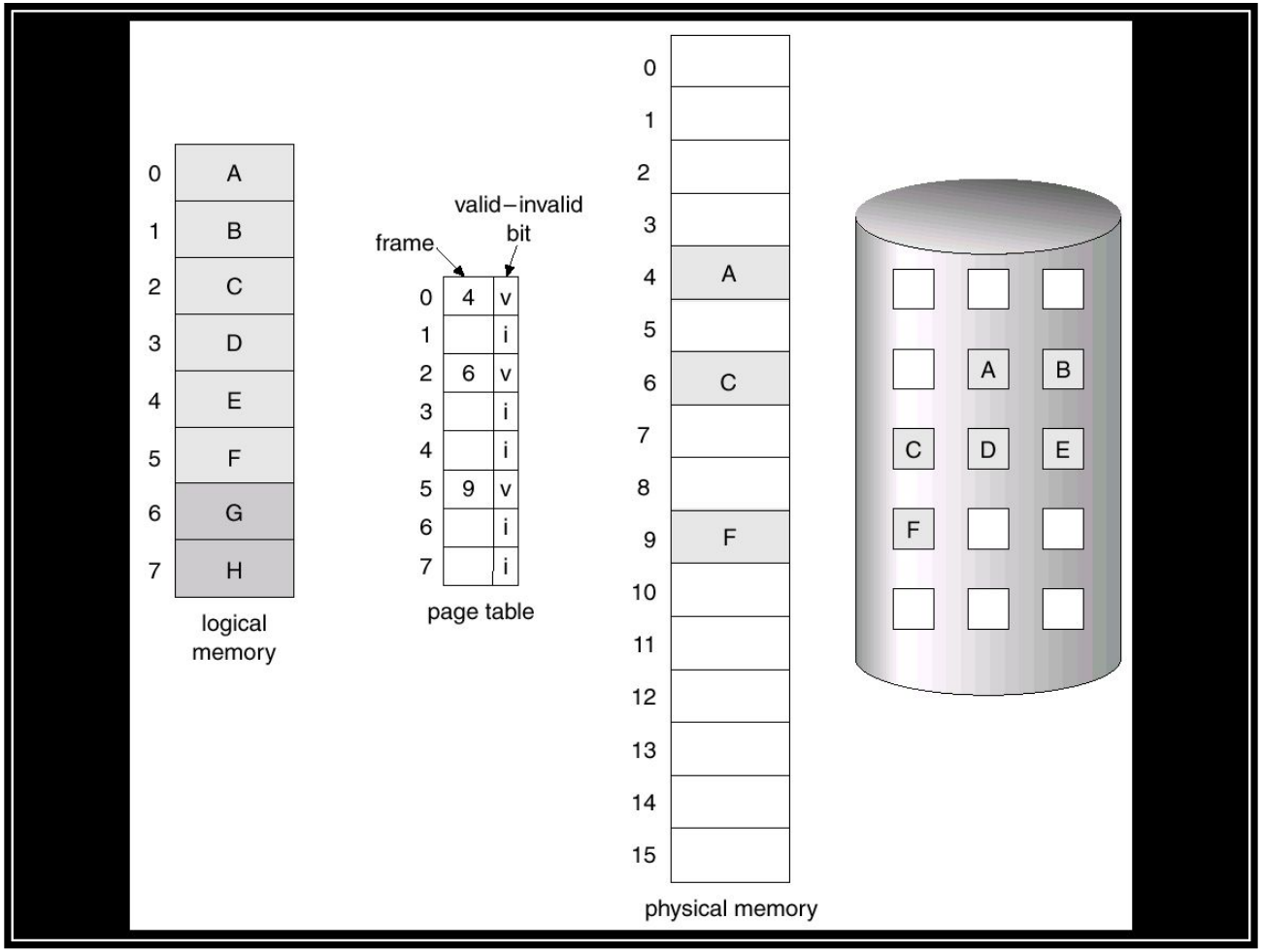
Виртуальная память больше, чем физическая память



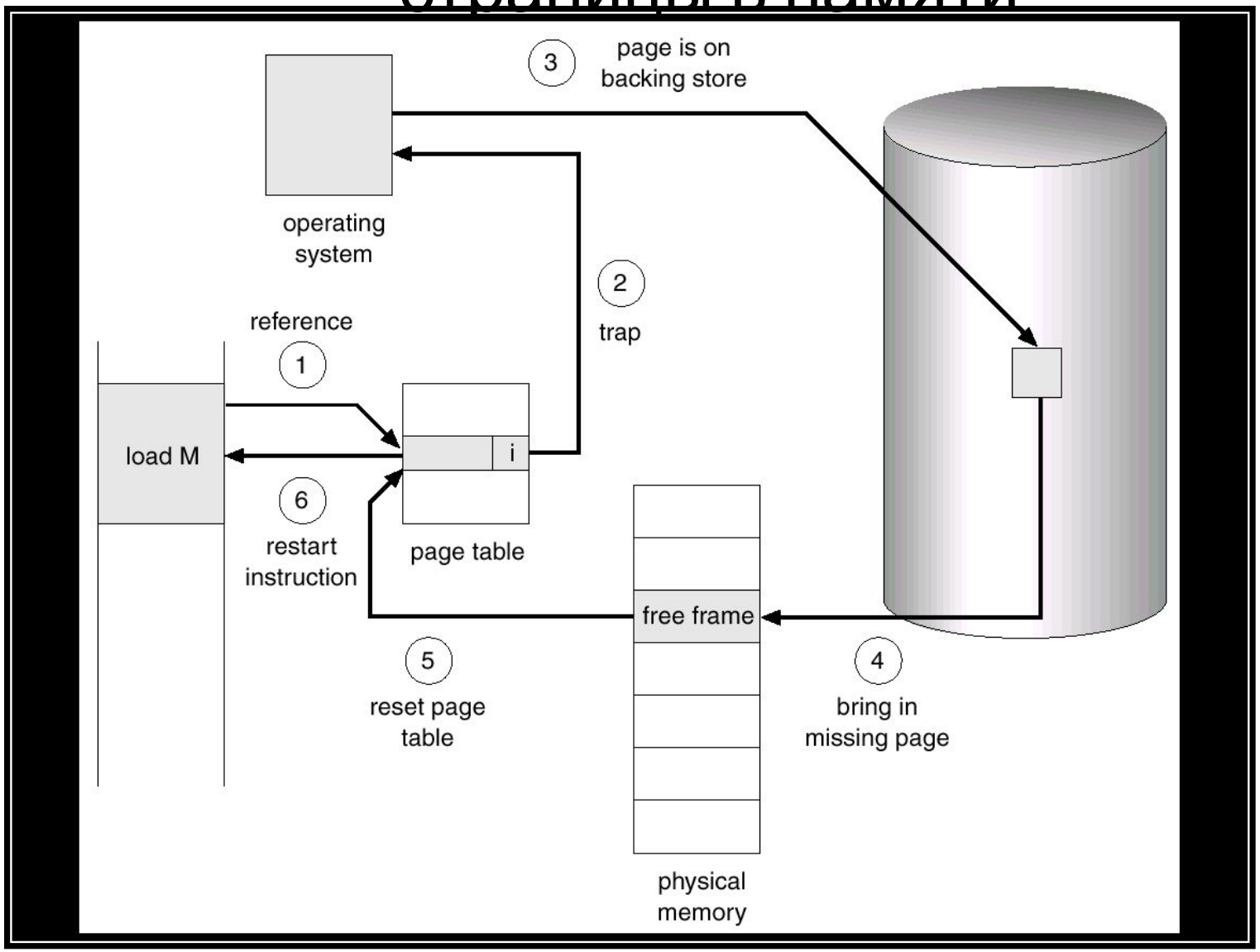
Преобразование страничной памяти в непрерывное дисковое пространство



Пример таблицы страниц, в которой не все страницы присутствуют в памяти



Этапы обработки ситуации отсутствия страницы в памяти



Оценка производительности стратегии обработки страницы по требованию

- **Коффициент отказов страниц (Page Fault Rate) $0 \leq p \leq 1.0$**
  - **Если $p = 0$ – отсутствие отказов страниц**
  - **Если $p = 1$, каждое обращение к странице приводит к отказу**

- **Эффективное время доступа (Effective Access Time - EAT)**

  EAT = (1 – p) * время доступа к памяти

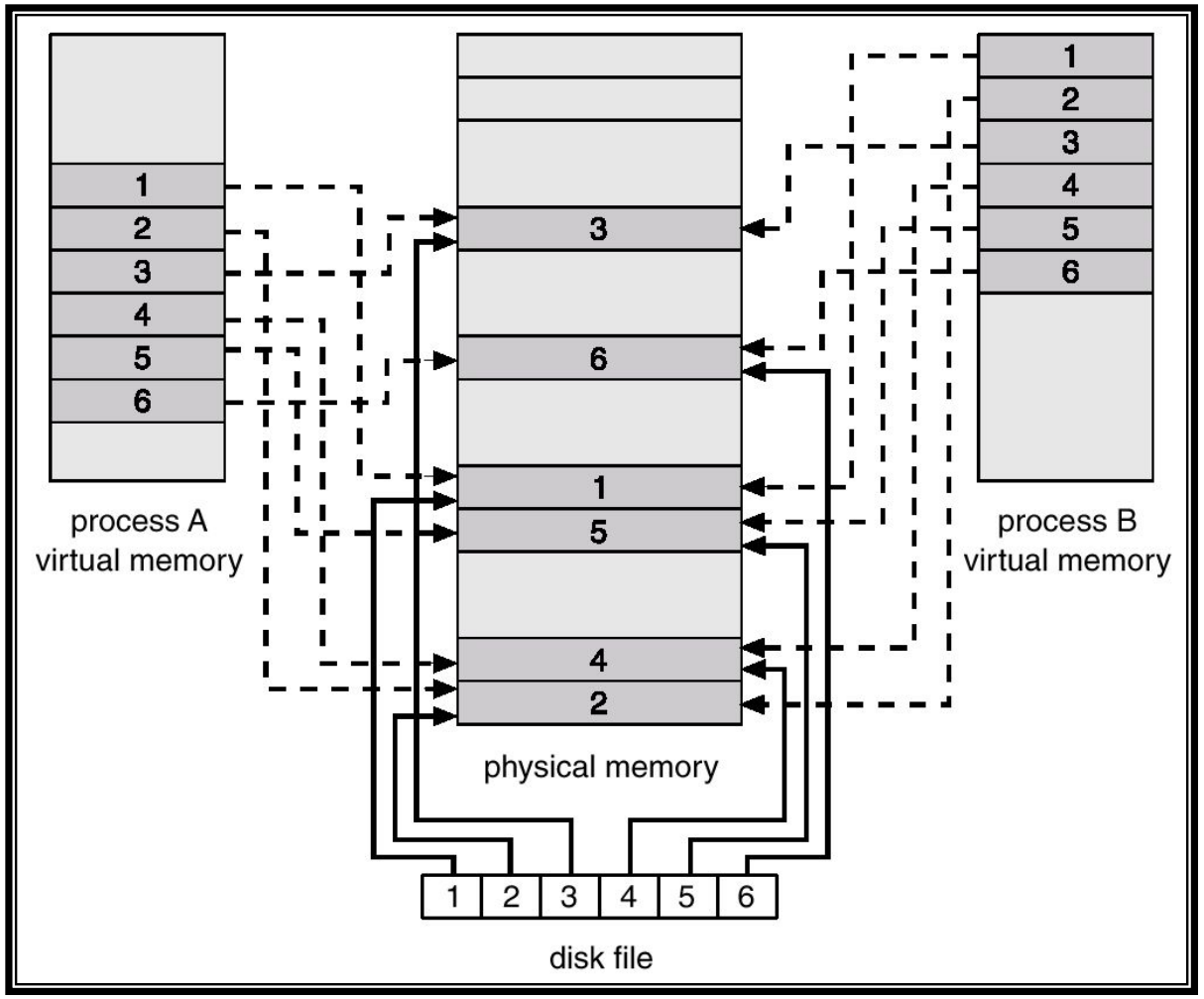
  + p * (время реакции на отказ

  + [время откачки страницы]

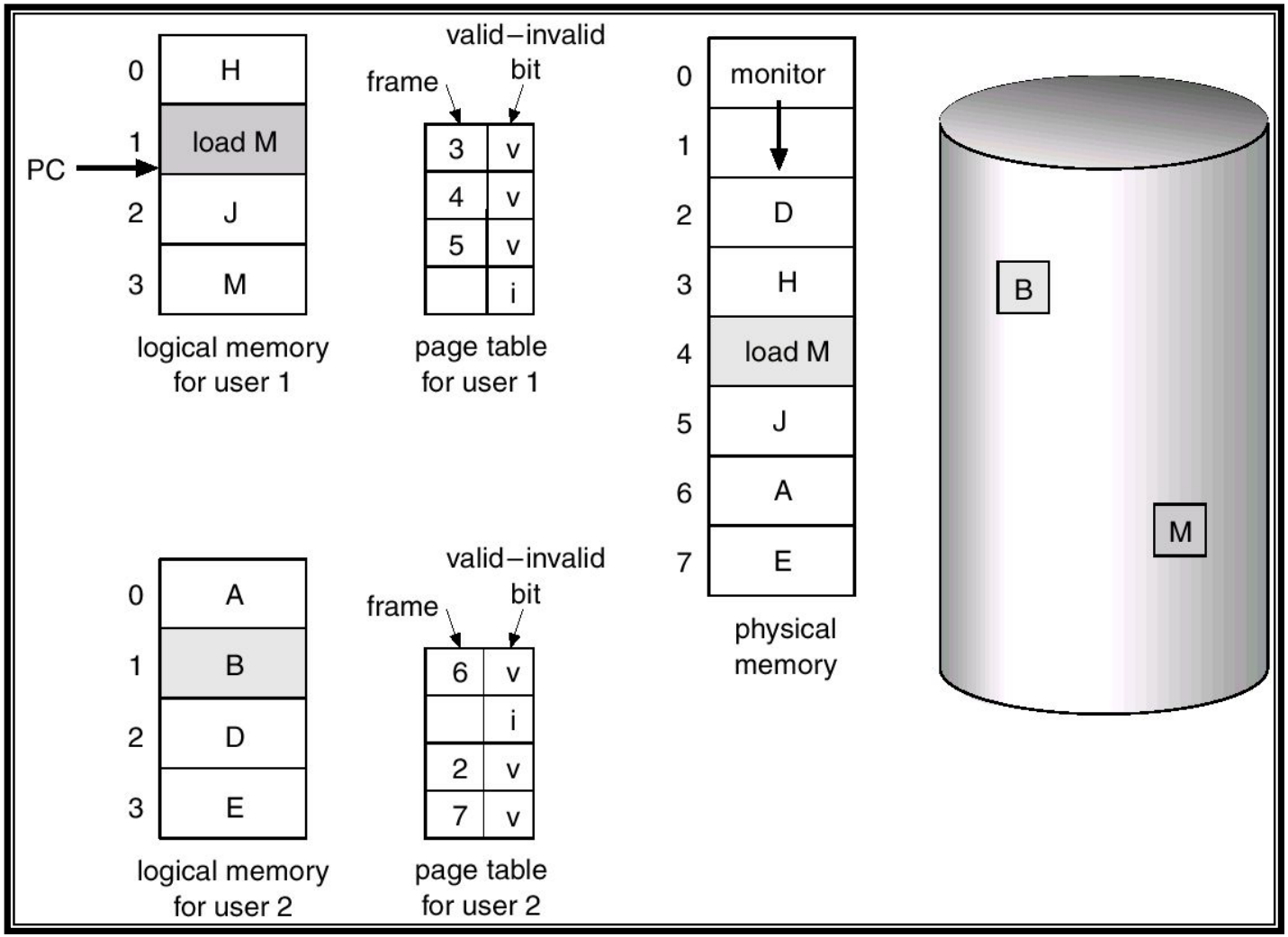
  + время подкачки страницы

  + время рестарта)

Файлы, отображаемые в память



Пример: замещение страниц



Замещение страниц

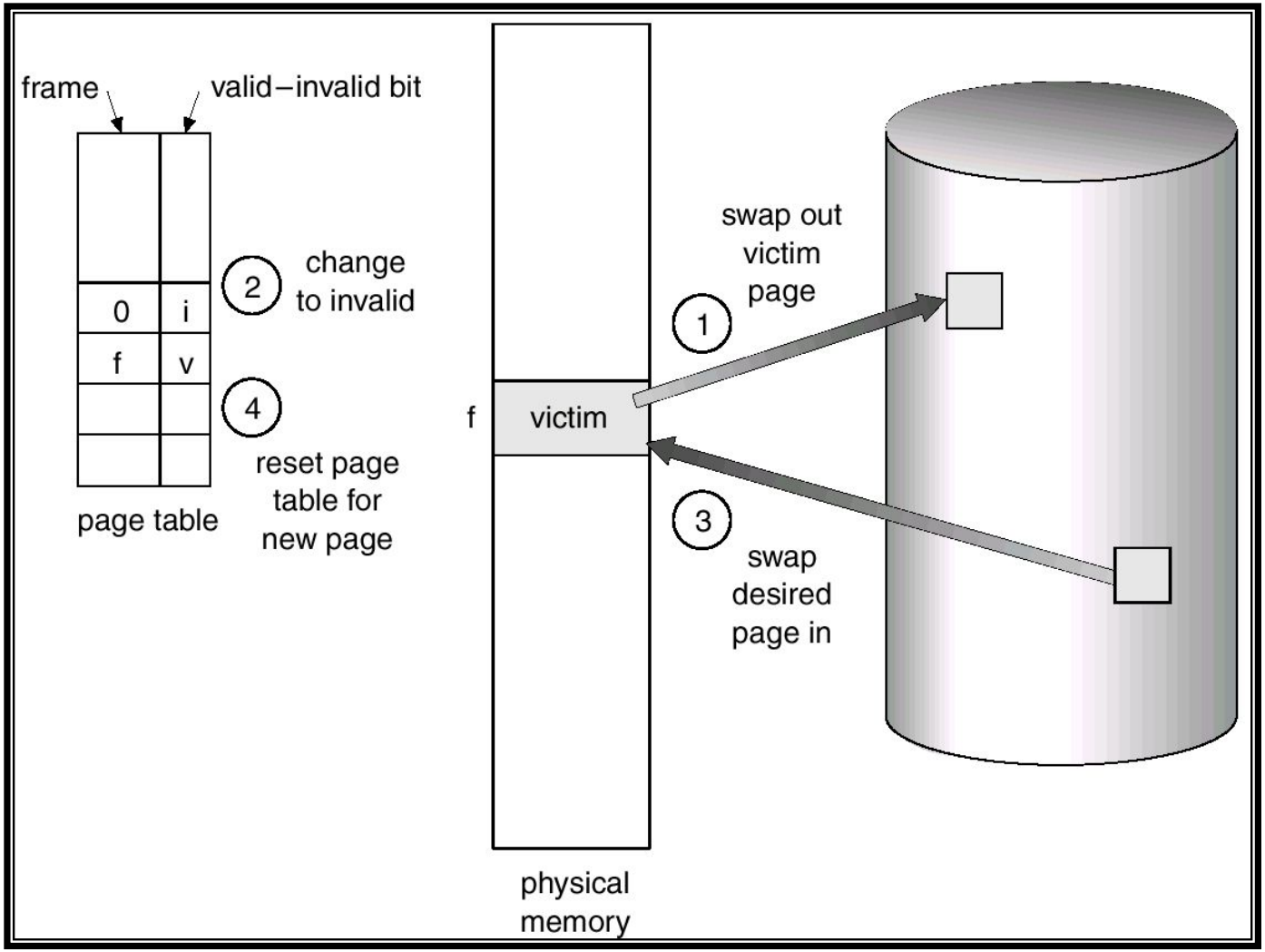
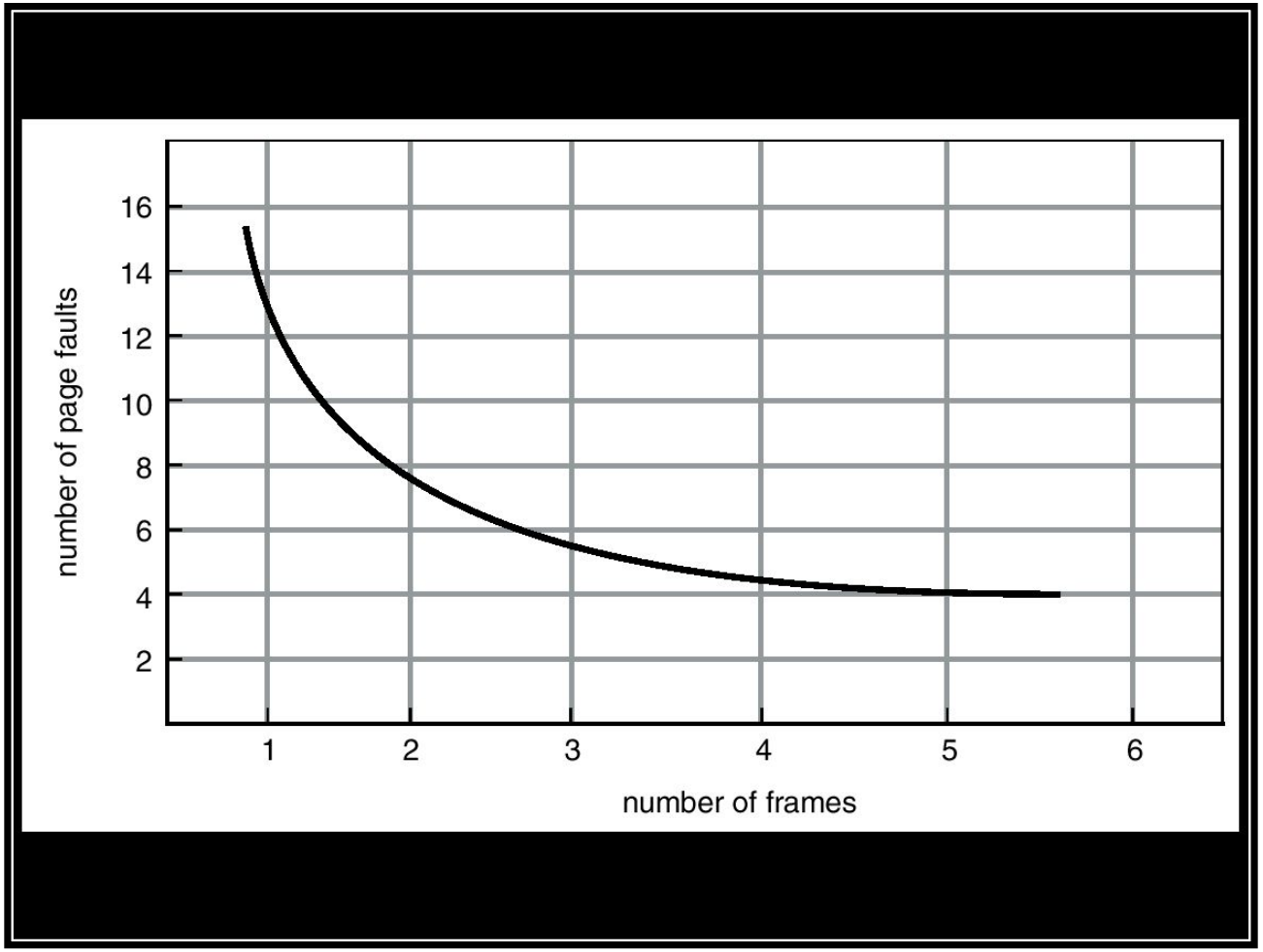


График зависимости числа отказов страниц от числа фреймов



Алгоритм FIFO (First-in-First-Out)

- **Наиболее простой алгоритм замещения страниц – в качестве жертвы всегда выбирается фрейм, первым из имеющихся считанный в основную память.**
- **Строка запросов: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**
- **3 фрейма (3 страницы могут быть одновременно в памяти для одного процесса)**

  (1, 2, 3) (4, 1, 2) (5, 3, 4)

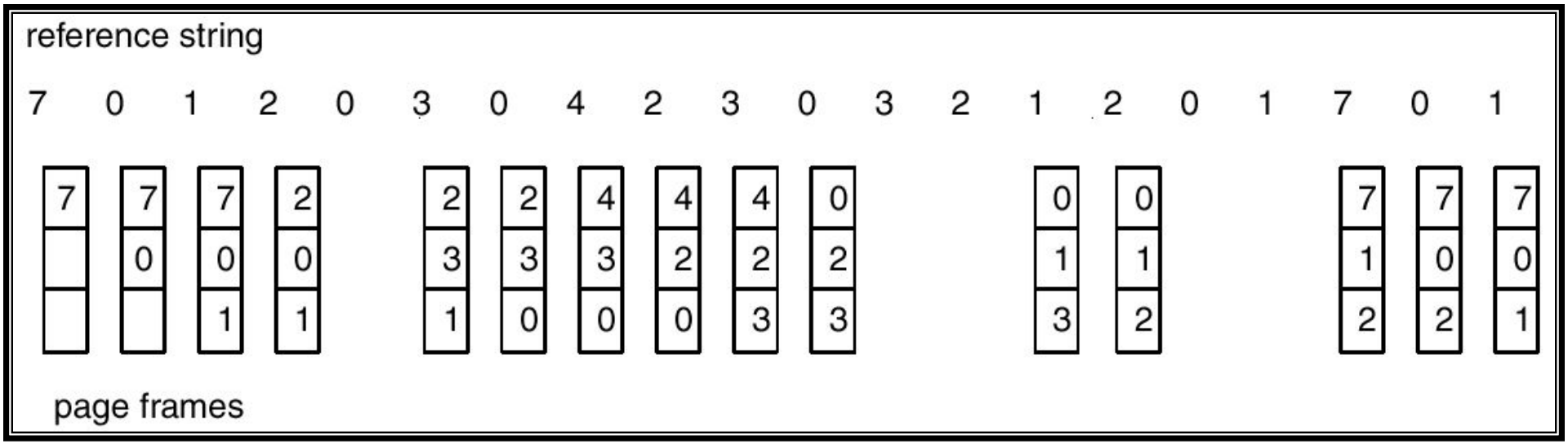
  9 отказов страниц
- **4 фрейма**

  (1, 2, 3, 4) (5, 1, 2, 3) (4, 5)

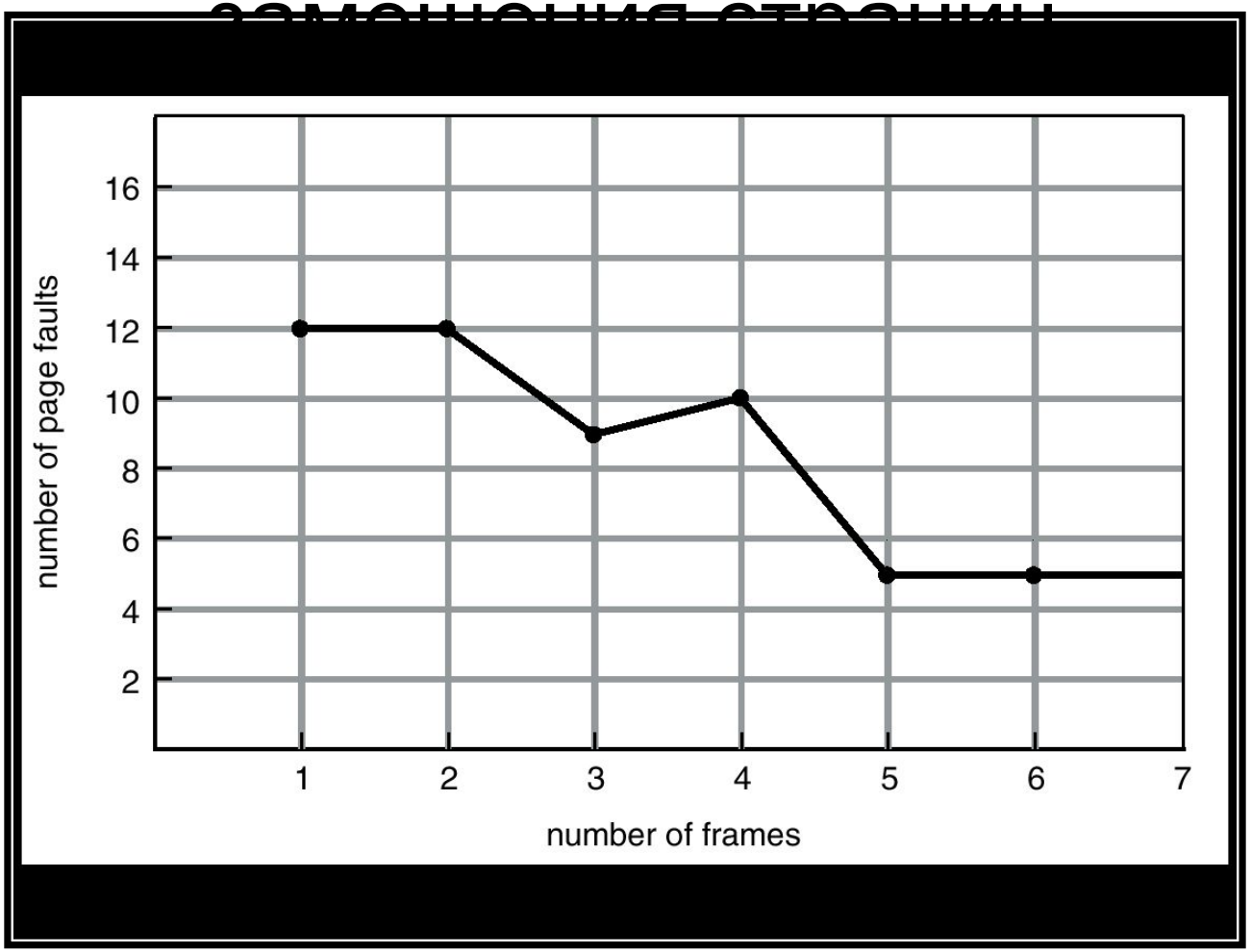
  10 (!) отказов страниц

- **FIFO – алгоритм замещения => аномалия Belady**
  - **больше фреймов ⇒ меньше отказов страниц**

Пример замещения страниц по алгоритму FIFO



Аномалия Belady при использовании алгоритма FIFO замещения страниц

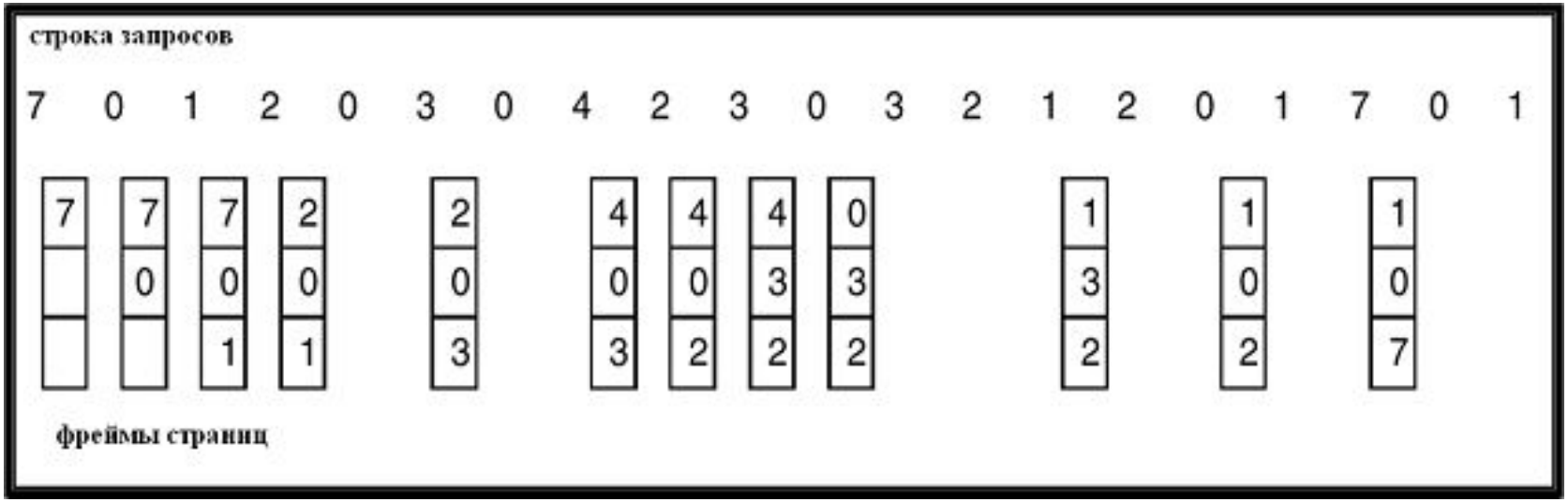


Алгоритм Least Recently Used (LRU)

- **Замещается та страница, которая раньше всего использовалась.**
- **Строка запросов: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**

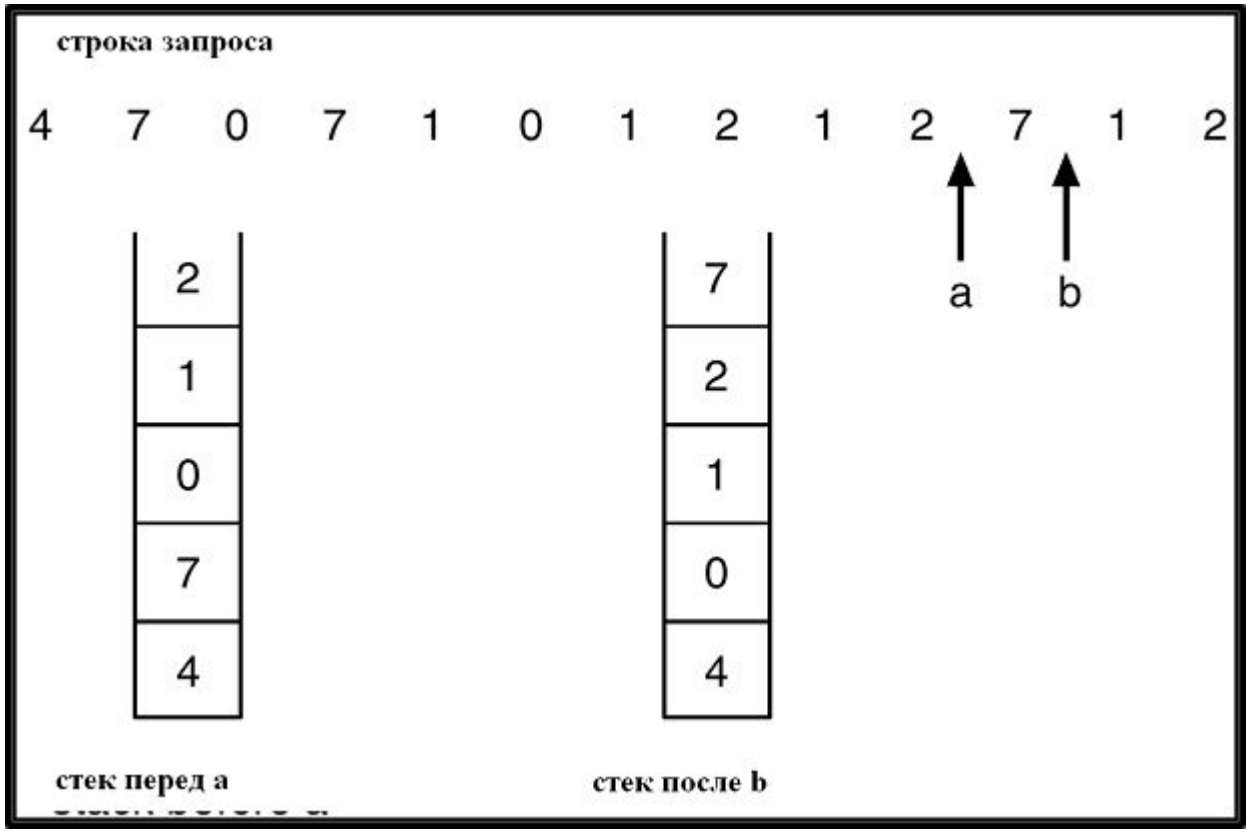
  (1 2 3 4)
  5 5 3
  4
- **Реализация счетчиков:**
  - **Каждый элемент таблицы страниц содержит счетчик; каждый раз при обращении к странице через некоторый элемент таблицы страниц содержимое часов (clock) копируется в его поле счетчика.**
  - **Когда требуется изменение в конфигурации страниц, необходимо проанализировать поля счетчиков, чтобы определить, какую именно страницу заместить (ту, у которой содержимое поля счетчика меньше => требуется применить алгоритм поиска минимального элемента в массиве; сложность O(n), где n – длина таблицы страниц)**

Замещение страниц по алгоритму LRU



Для оптимизации данного алгоритма, чтобы избежать поиска минимального элемента таблицы страниц при каждом замещении страниц, используется **стековая реализация** – стек номеров страниц хранится в форме двухсвязного списка. При обращении к странице она перемещается в начало списка (для этого требуется изменить 6 указателей). Преимущества данной модификации алгоритма в том, что при замещении страниц не требуется поиска.

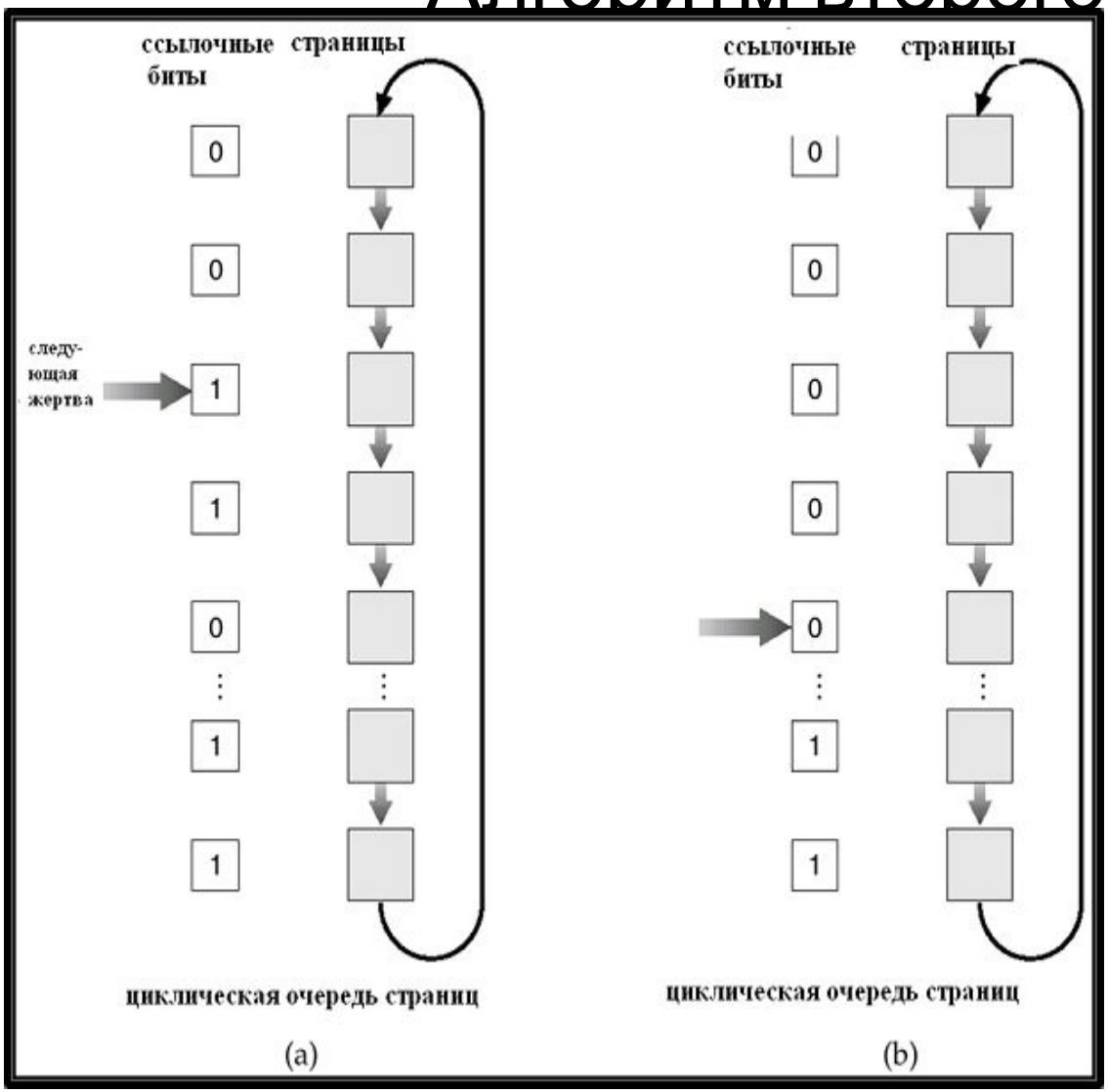
Использование стека для хранения информации о самых недавних обращениях к страницам



Алгоритмы, близкие к LRU

- Имеется несколько алгоритмов, близких к алгоритму LRU, в которых реализованы различные идеи улучшений или упрощений, направленные на то, чтобы уменьшить недостатки LRU.
- **Бит ссылки (reference bit)**.В данном алгоритме с каждой страницей связывается бит, первоначально равный 0. При обращении к странице бит устанавливается в 1. Далее, при необходимости замещения страниц, заменяется та страница, у которой бит равен 0 (если такая существует), т.е. страница, к которой не было обращений. Данная версия алгоритма позволяет избежать поиска по таблице страниц. Однако она, очевидно, менее оптимальна, чем LRU.
- **Второй шанс (second chance)**.В данной версии алгоритма используются ссылочный бит и показания часов, которые хранятся в каждом элементе таблицы страниц. Замещение страниц основано на показаниях часов. Если страница, которую следует заместить (по показаниям часов), имеет ссылочный бит, равный 1, то выполняются следующие действия:
  - Установить ссылочный бит в 0;
  - Оставить страницу в памяти;
  - Заместить **следующую** страницу (по показаниям часов), по тем же самым правилам.

Алгоритм второго шанса



Данный алгоритм имеет след эвристическое обоснование. Странице, кот дольше всего не исп, ей как бы дается второй шанс на то, что она будет использована, т. е. делается предполож, что, по мере возраст времени, вероятность обращения к странице, к кот давно не было обращ, возрастает.

Фиксированное выделение

- **Равномерное распределение – например, если имеется 100 фреймов и 5 процессов, каждому выделяется по 20 страниц.**
- **Пропорциональное распределение – выделять фреймы в соответствии со следующим принципом:**если общее число фреймов m, размер процесса – s, а общий размер всех процессов – S, то общее число фреймов, выделенных процессу, равно:a = m * (s / S).

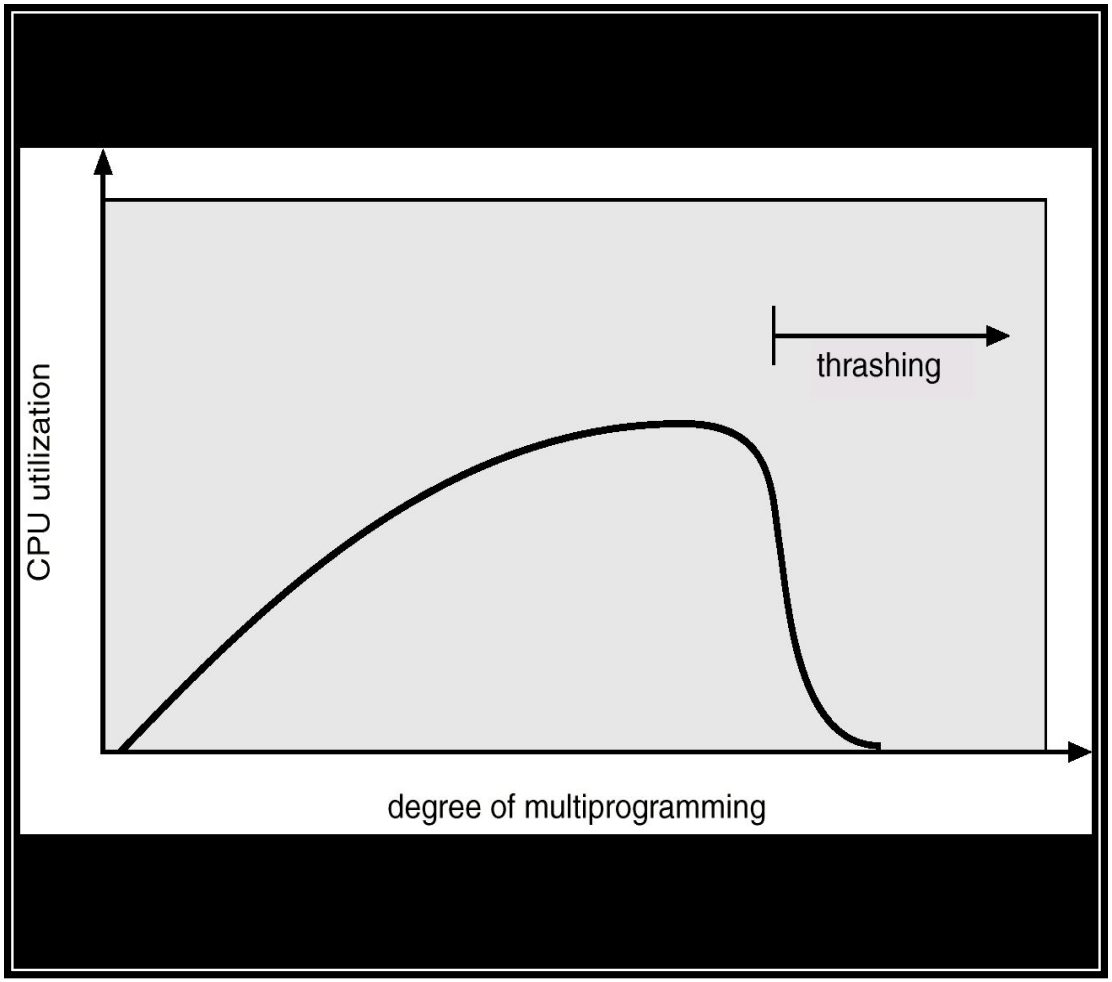
$$s_i = \text{size of process } p_i$$

$$S = \sum s_i$$

$$m = \text{total number of frames}$$

$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

Thrashing



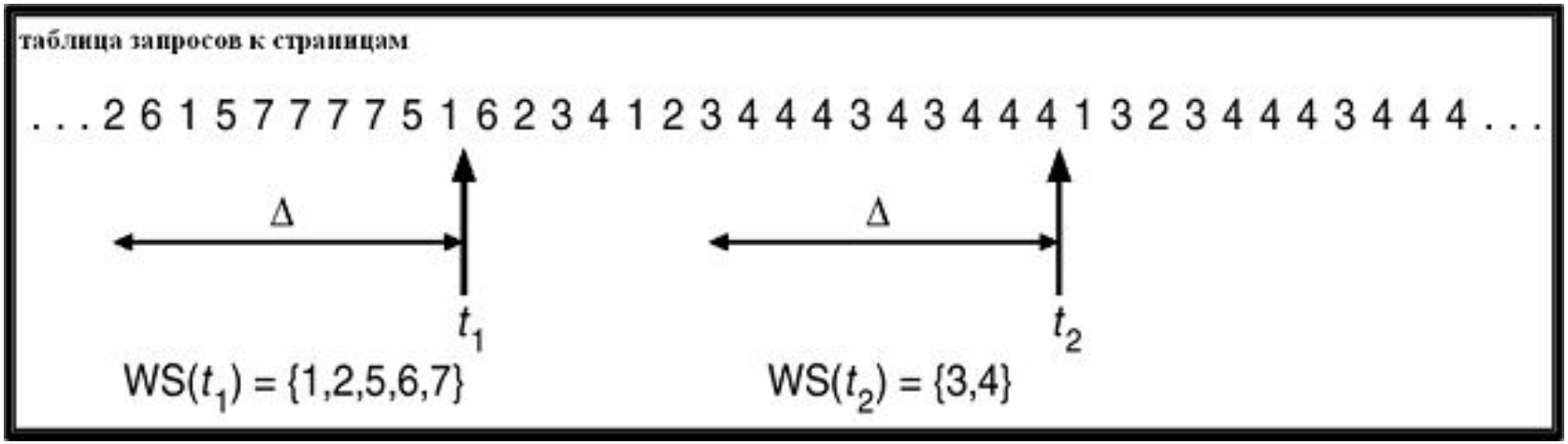
Если процессу не выделено достаточное число страниц, коэффициент отказов страниц очень высок. Это приводит к тому, что процесс занят в основном откачкой и подкачкой страниц.
thrashing означает катастрофическую нехватку фреймов в основной памяти. На практике для пользователя это выглядит следующим образом. При очень большом числе обрабатываемых процессов полезность использования процессора резко падает из-за постоянных откачек и подкачек. Это и есть **thrashing**.

Модель рабочего множества

- **thrashing происходит, если сумма размеров локальных потребностей процессов в основной памяти больше общего размера памяти.Для борьбы с подобными явлениями в операционных системах для распределения фреймов используется модельрабочего множества.**
- **Δ ≡ рабочее множество ≡ фиксированное число обращений к страницам**

- ***WSS*$_i$ (рабочее множество процесса *P*$_i$) = общее число обращений к страницам в самой недавней Δ (меняется в зависимости от времени)**
  - **если Δ очень мало, не рассматриваем полную локальную потребность.**
  - **Если Δ слишком велико, рассматриваем несколько локальных потребностей.**
  - **Если Δ = ∞ ⇒ рассматриваем всю программу.**
- ***D* = Σ *WSS*$_i$ ≡ общий объем требований фреймов**
- **Если *D* > *m* ⇒ Thrashing (*m* - общий размер памяти)**
- **Политика ОС: если *D* > m, приостановить один из процессов.**

Модель рабочего множества



Атрибуты файлов

- **Имя (Name) – информация в символьной форме, воспринимаемая человеком.**
- **Тип (Type) – необходим для систем, которые поддерживают различные типы файлов (Эльбрус: тип файла – *число*; 0 – данные, 2 – код, 3 – текст и т.д.). В MS DOS, Windows, UNIX тип файла приняно кодировать расширением имени.**
- **Размещение (Location) – указатель на размещение файла на устройстве.**
- **Размер (Size) – текущий размер файла.**
- **Защита (Protection) – управляющая информация о том, кто может читать, изменять и исполнять файл.**
- **Время, дата, идентификация пользователя.**
- **Информация о файлах хранится в структуре директорий.**

Операции над файлом

- **Создание - Create**
- **Запись - Write**
- **Чтение - Read**
- **Поиск позиции внутри файла - Seek**
- **Удаление - Delete**
- **Сокращение - Truncate**
- **Open(F_i) – поиск в структуре директорий на диске элемента F_i, и перемещение содержимого элемента в память.**
- **Close (F_i) – переместить содержимое элемента F_i из памяти в структуру директорий на диске.**

Типы файлов – имя и расширение

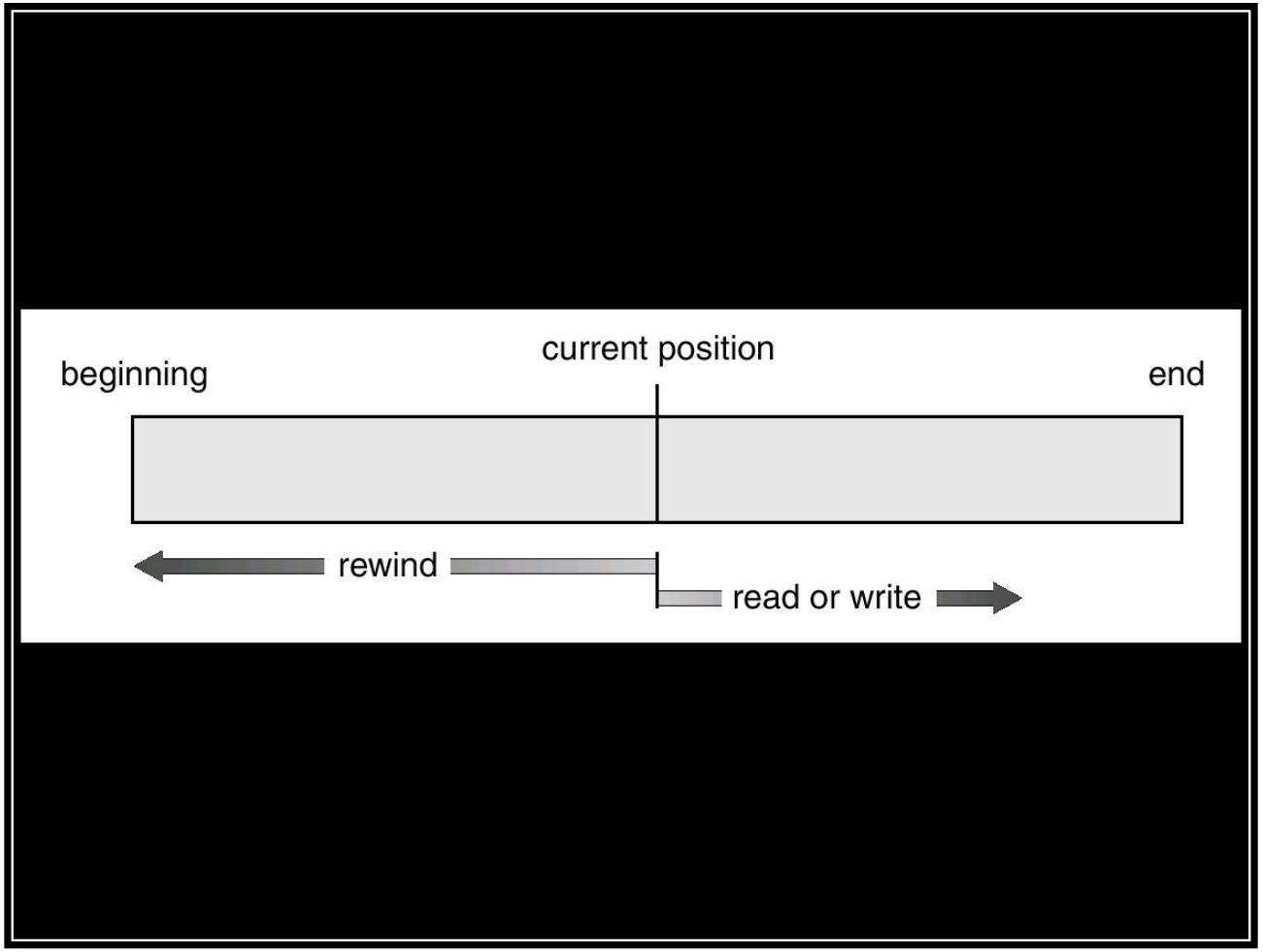
file type	usual extension	function
executable	exe, com, bin or none	read to run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rrf, doc	various word-processor formats
library	lib, a, so, dll, mpeg, mov, rm	libraries of routines for programmers
print or view	arc, zip, tar	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm	binary file containing audio or A/V information

Методы доступа к файлам

- **Последовательный доступ**
  read next
  write next
  reset
  rewrite
- **Прямой доступ**
  read n
  write n
  position to n
  read next
  write next
  rewrite n

***n* = относительный номер блока**

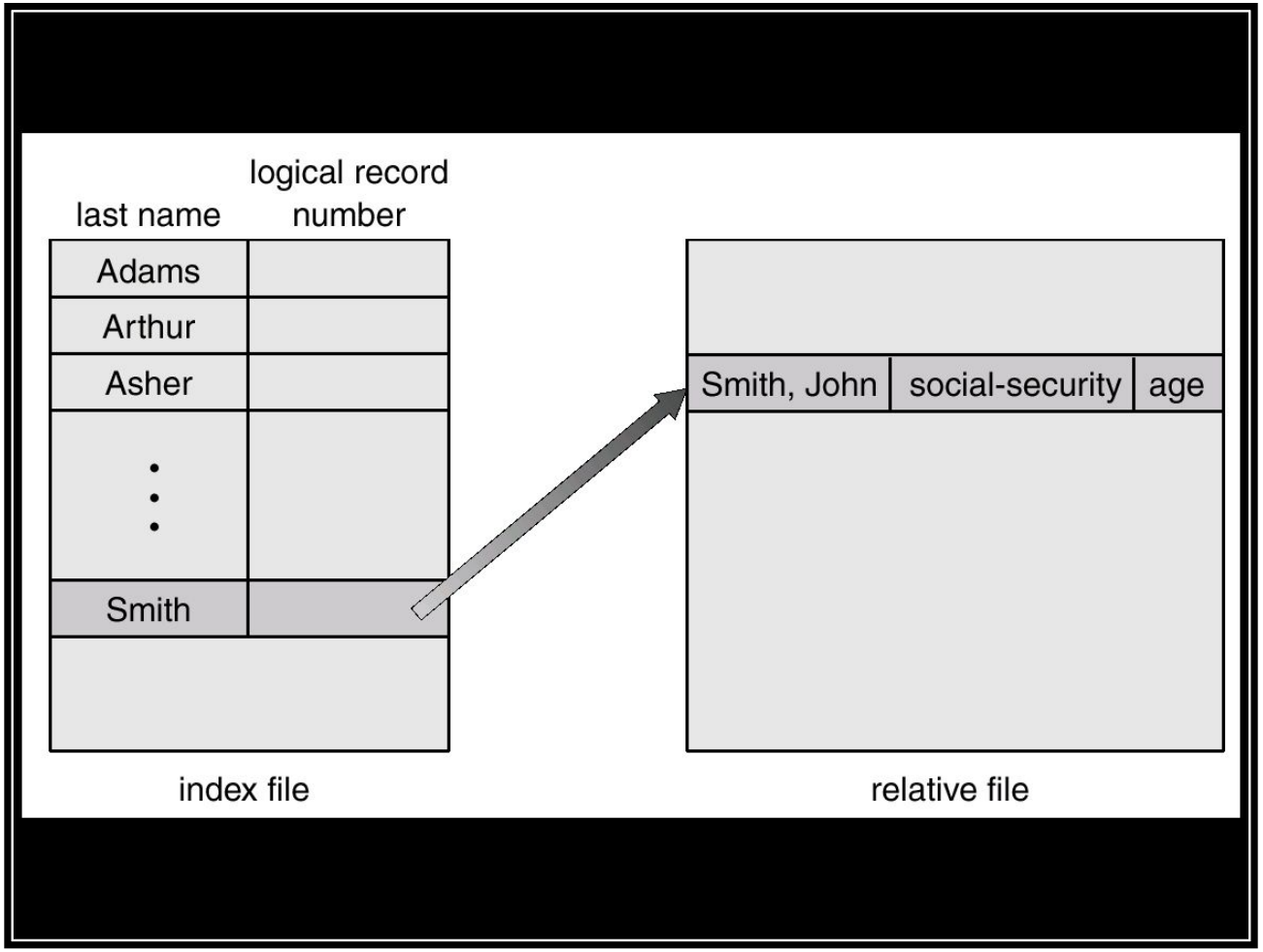
Файл последовательного доступа



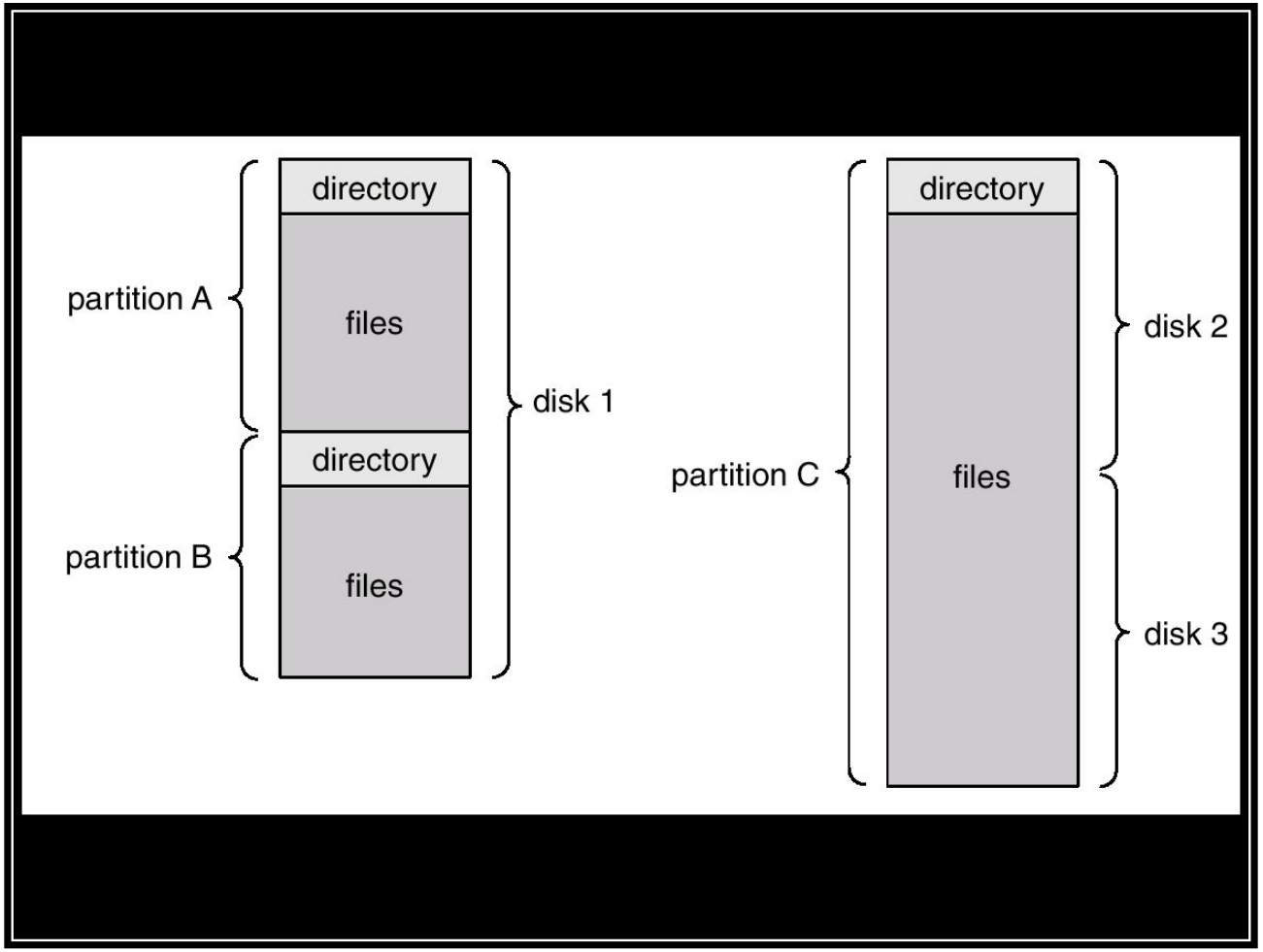
Моделирование последовательного доступа для файла с прямым доступом

sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp*+1;
write next	*write cp*; *cp* = *cp*+1;

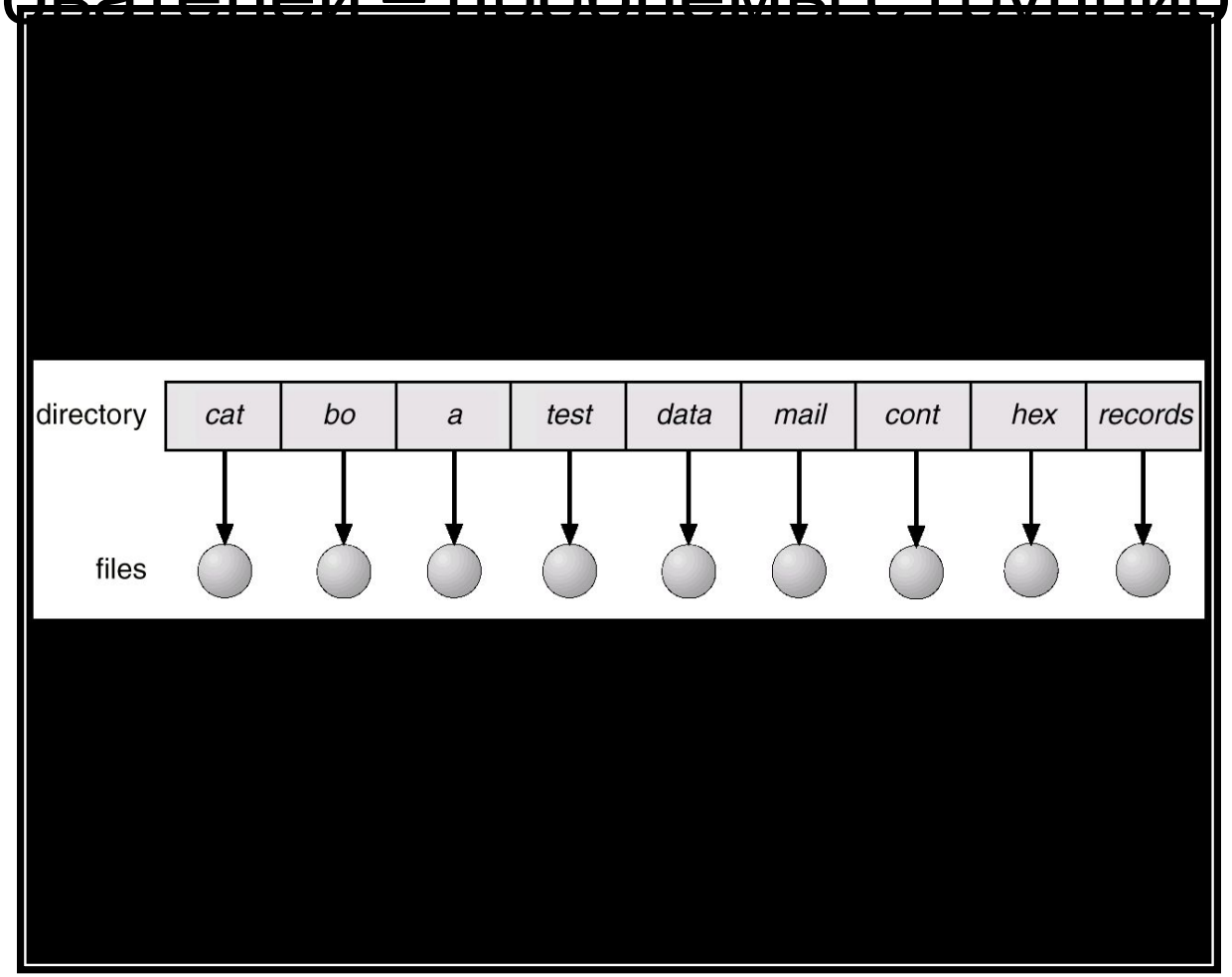
Пример индексного файла и файла, представляющего отношение (relative file)



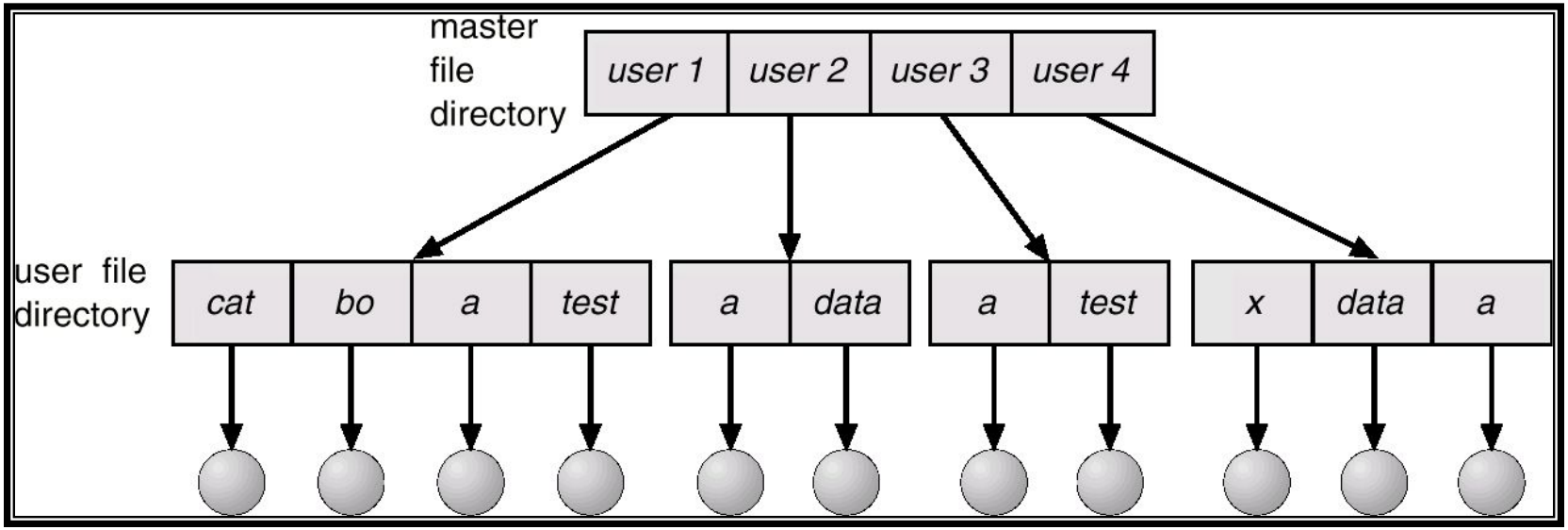
Типичная организация файловой системы



Одноуровневая организация для всех пользователей – проблемы с группировкой

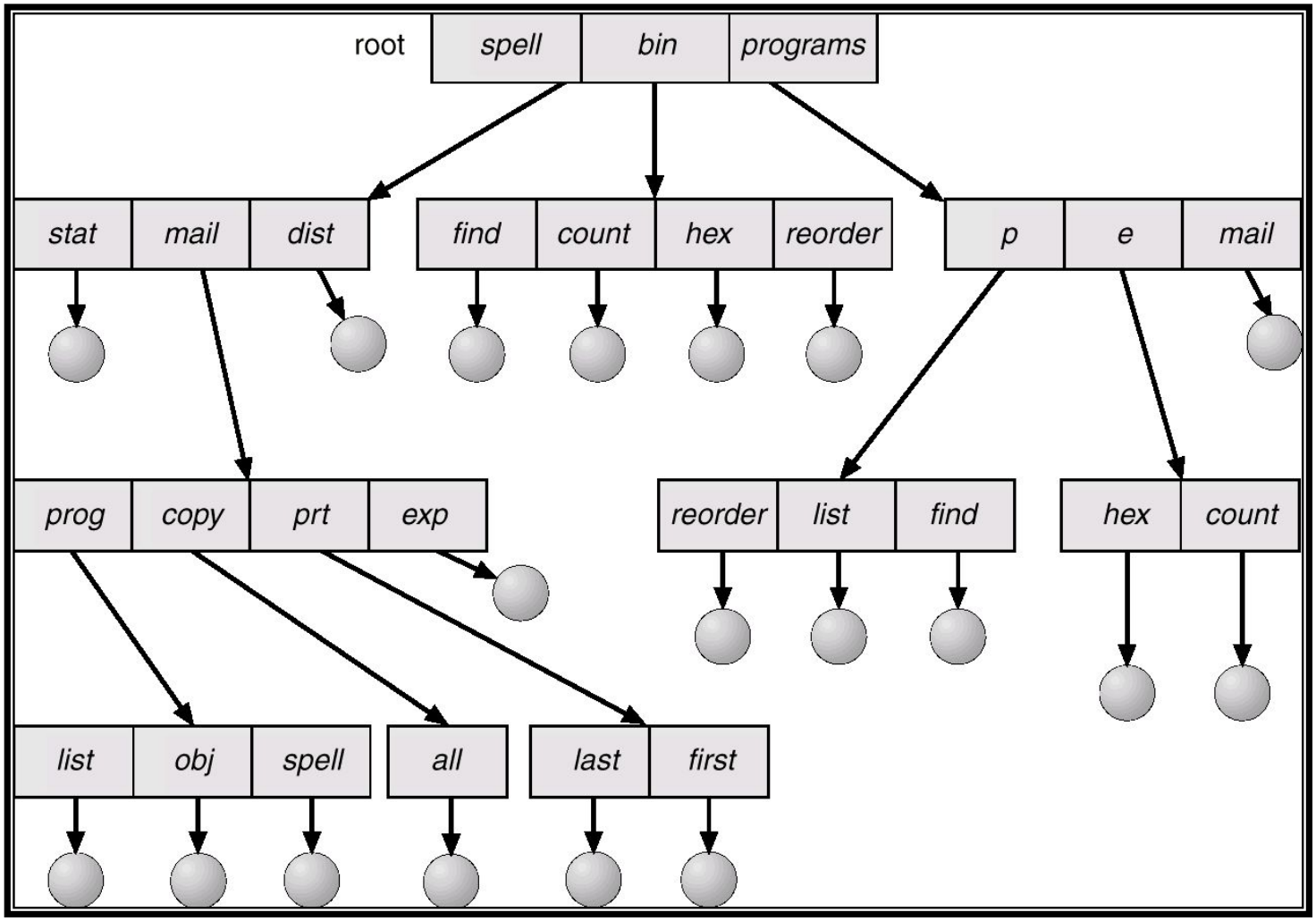


Двухуровневая организация

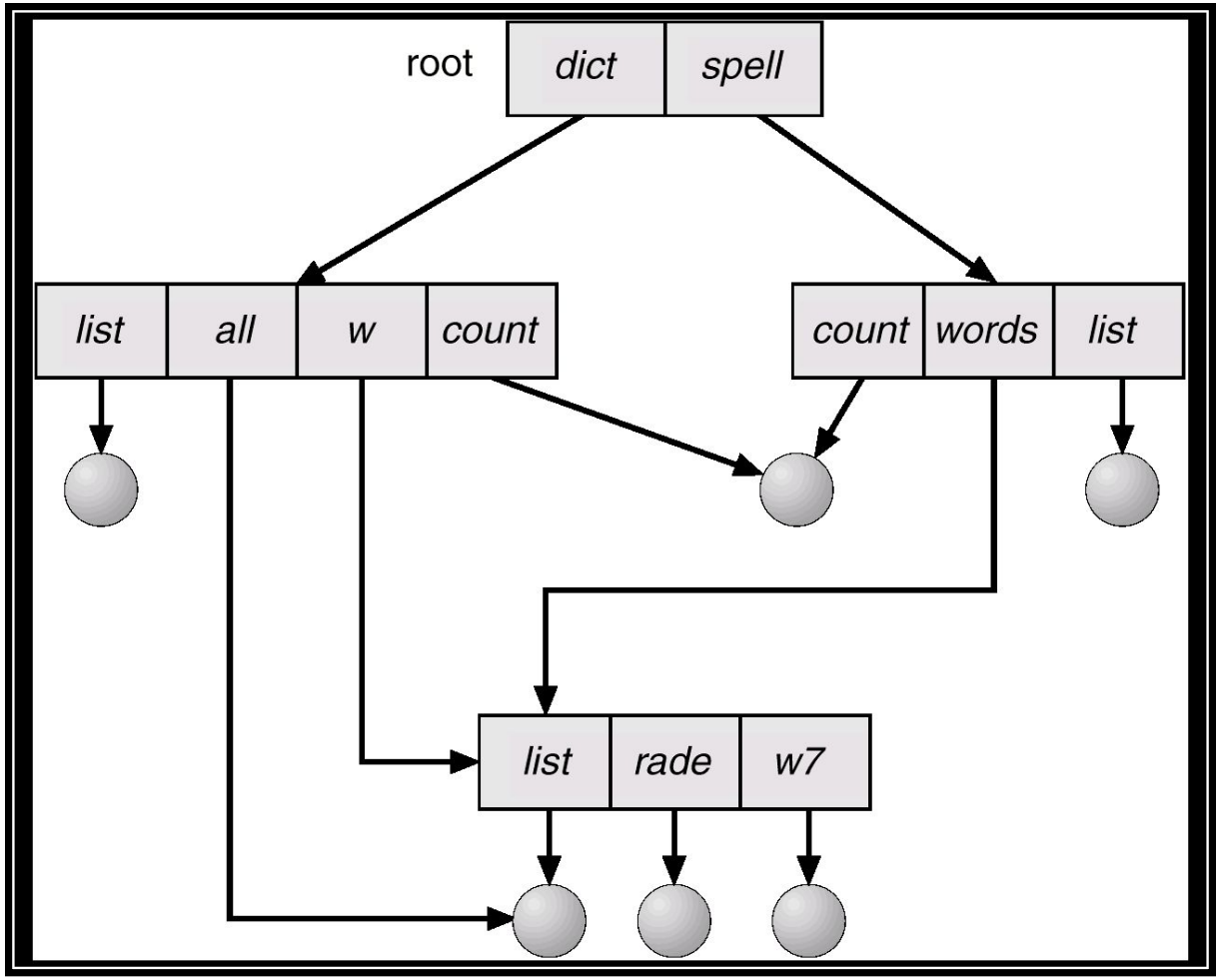


- Имя пути
- Возможность иметь одинаковые имена файлов для различных пользователей
- Эффективный поиск
- Нет возможности группировки

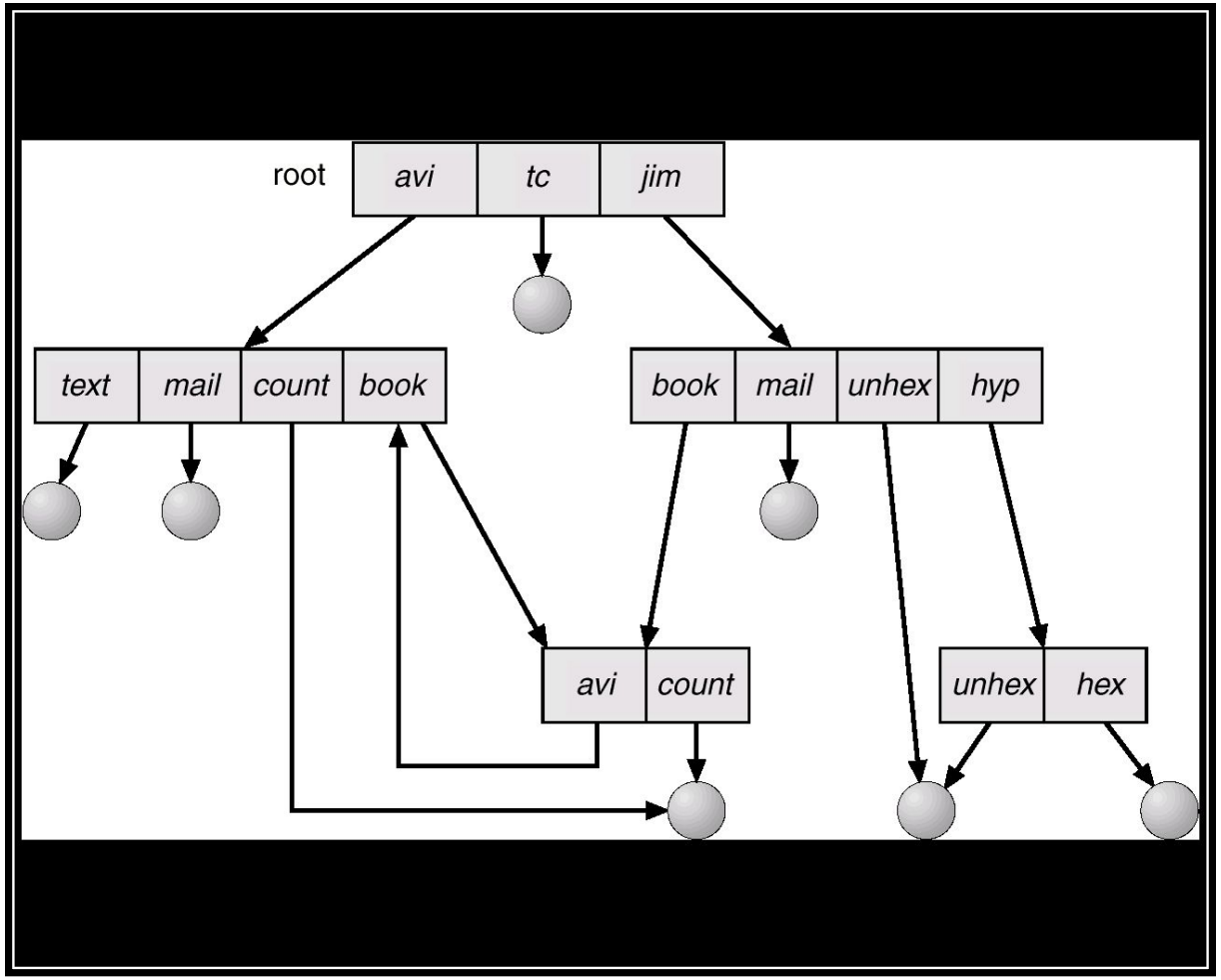
Древовидная структура директорий



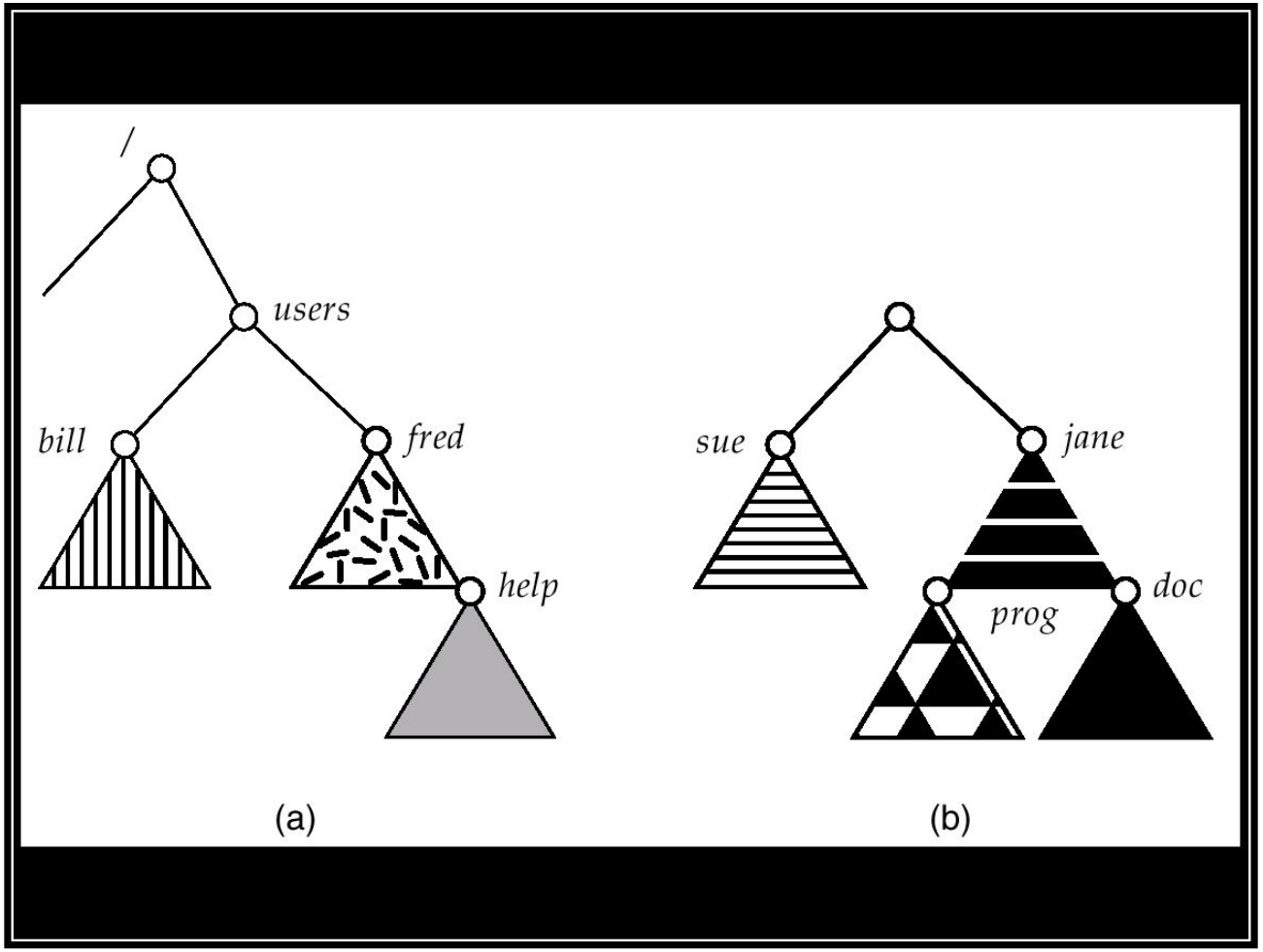
Структура директорий в виде ациклического графа (с разделяемыми директориями и файлами)



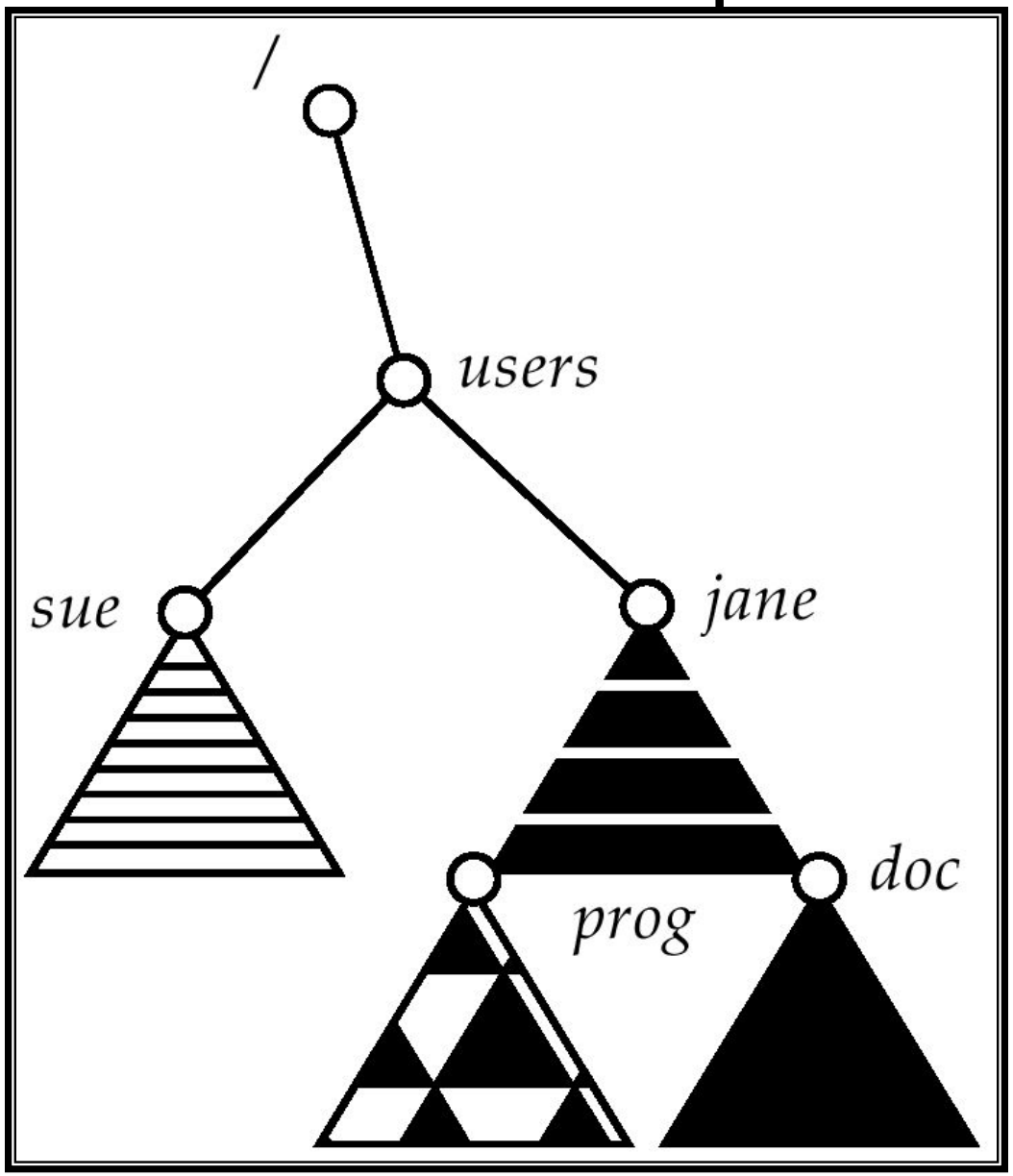
Структура директорий в виде произвольного графа



Дерево смонтированных систем (a) и еще не смонтированная файловая система (b)



Точка монтирования



Защита (protection)

- **Создатель файла должен иметь возможность управлять:**
  - **Списком допустимых операций над файлом**
  - **Списком пользователей, которым они разрешены**
- **Типы доступа:**
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**

Списки доступа и группы (UNIX)

- **Режимы доступа: read, write, execute**
- **Три класса пользователей:**

  RWX

  a) owner access 7 ⇒ 1 1 1

  RWX

  b) group access 6 ⇒ 1 1 0

  RWX

  c) public access 1 ⇒ 0 0 1
- **Системный администратор создает группу (например, JAVA) и включает в нее нескольких пользователей.**
- **Для конкретного файла (например, *game*) или поддиректории определяются соответствующие полномочия доступа**
- **Для директории “X” означает возможность входа в нее (*cd)***

Типичная структура блока управления файлом

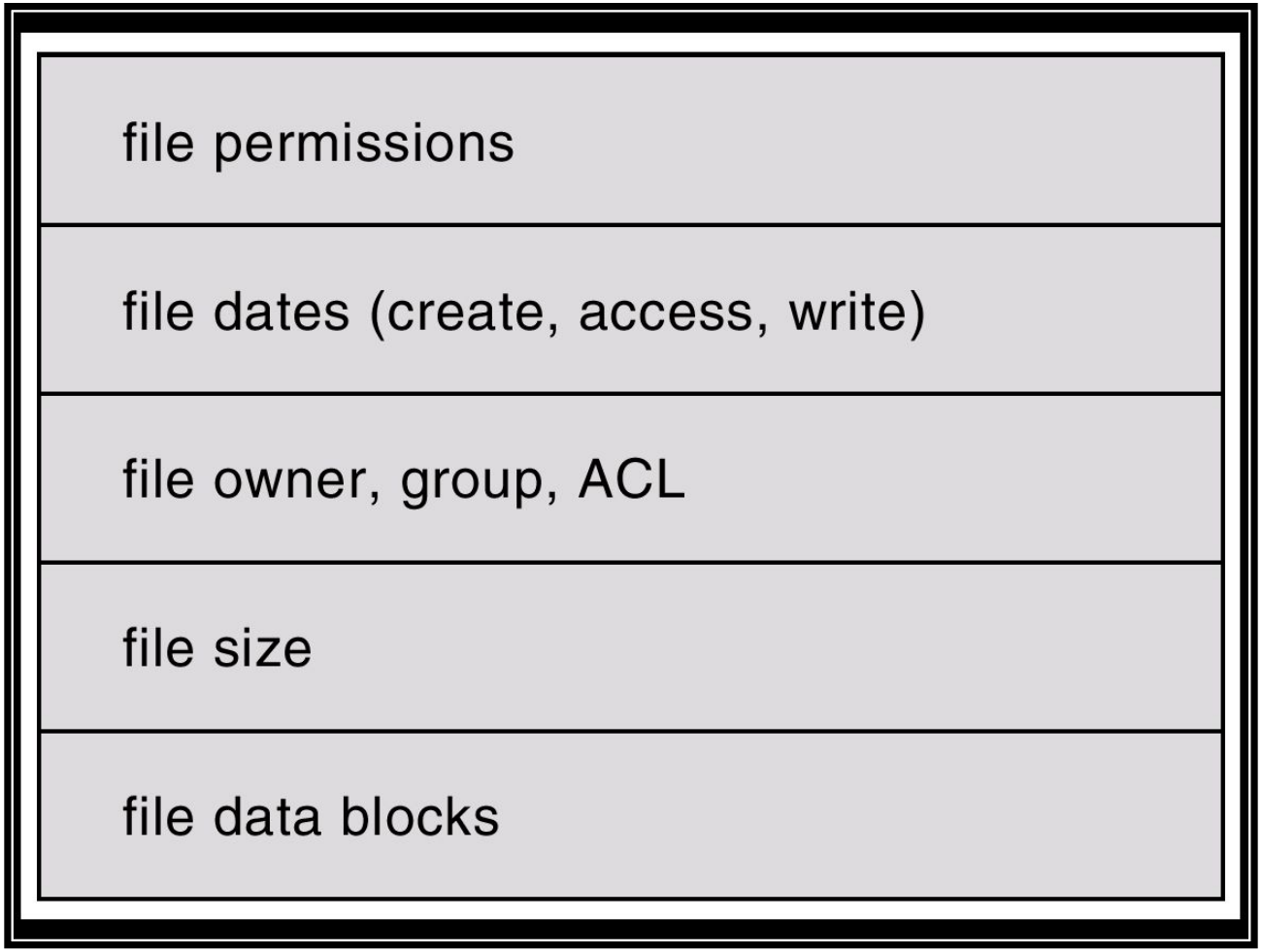
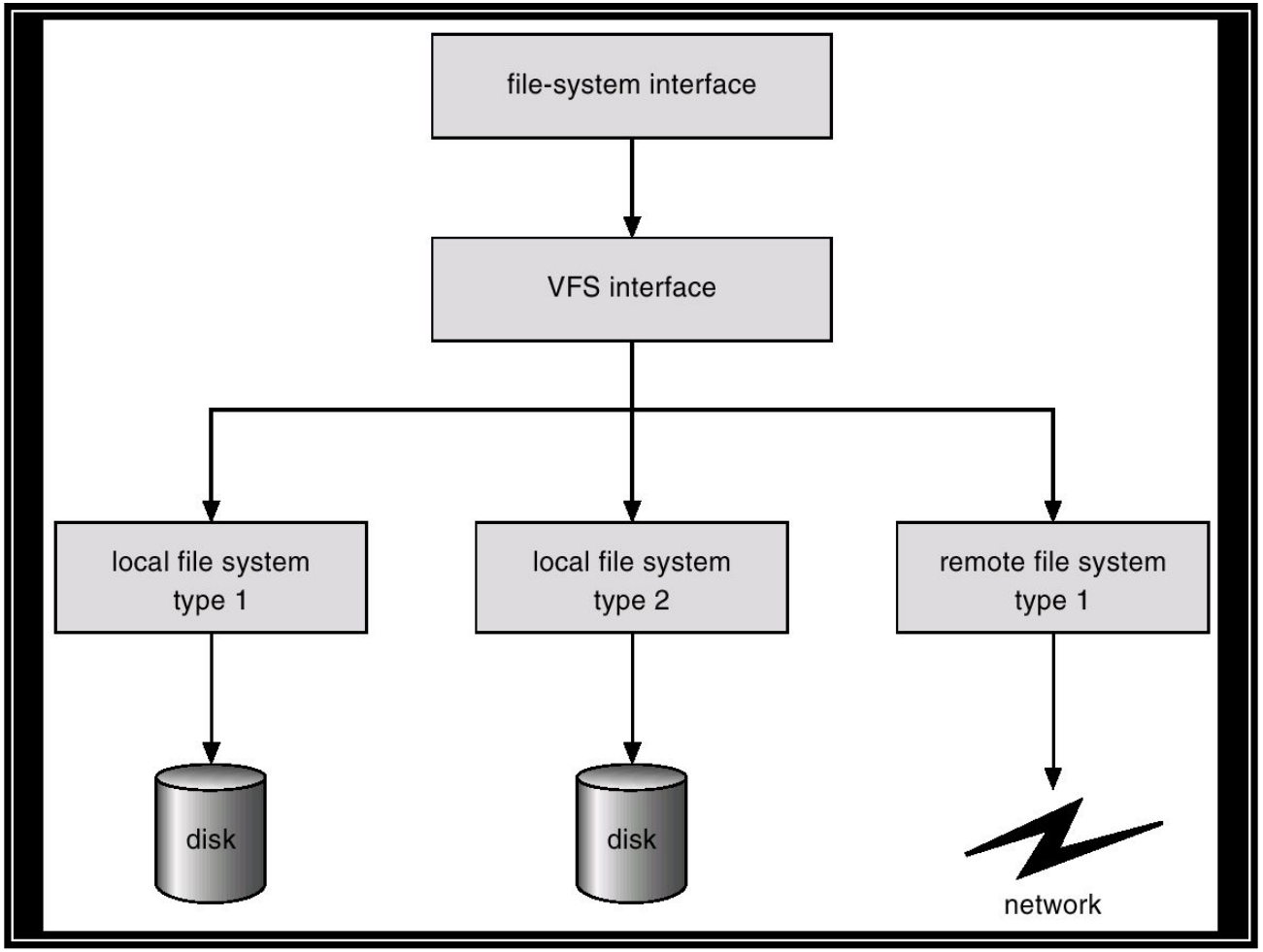
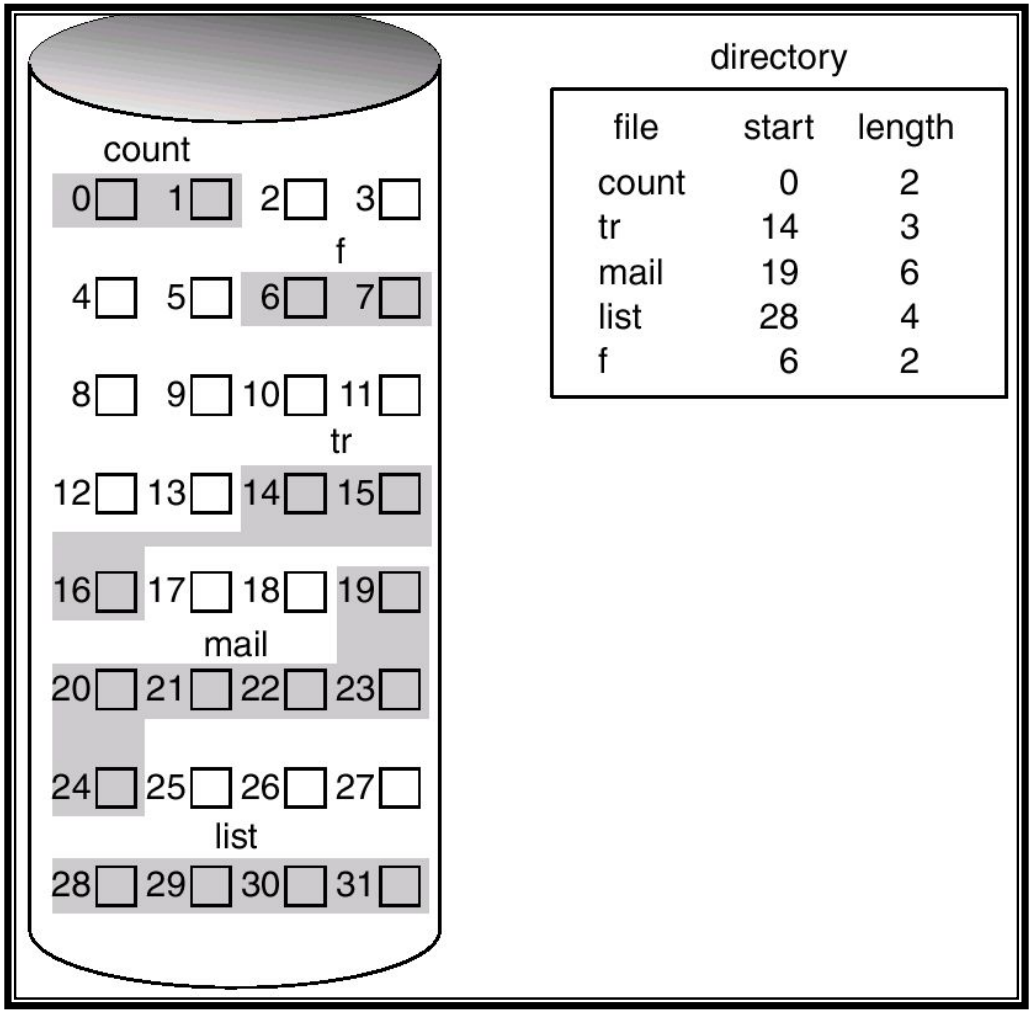


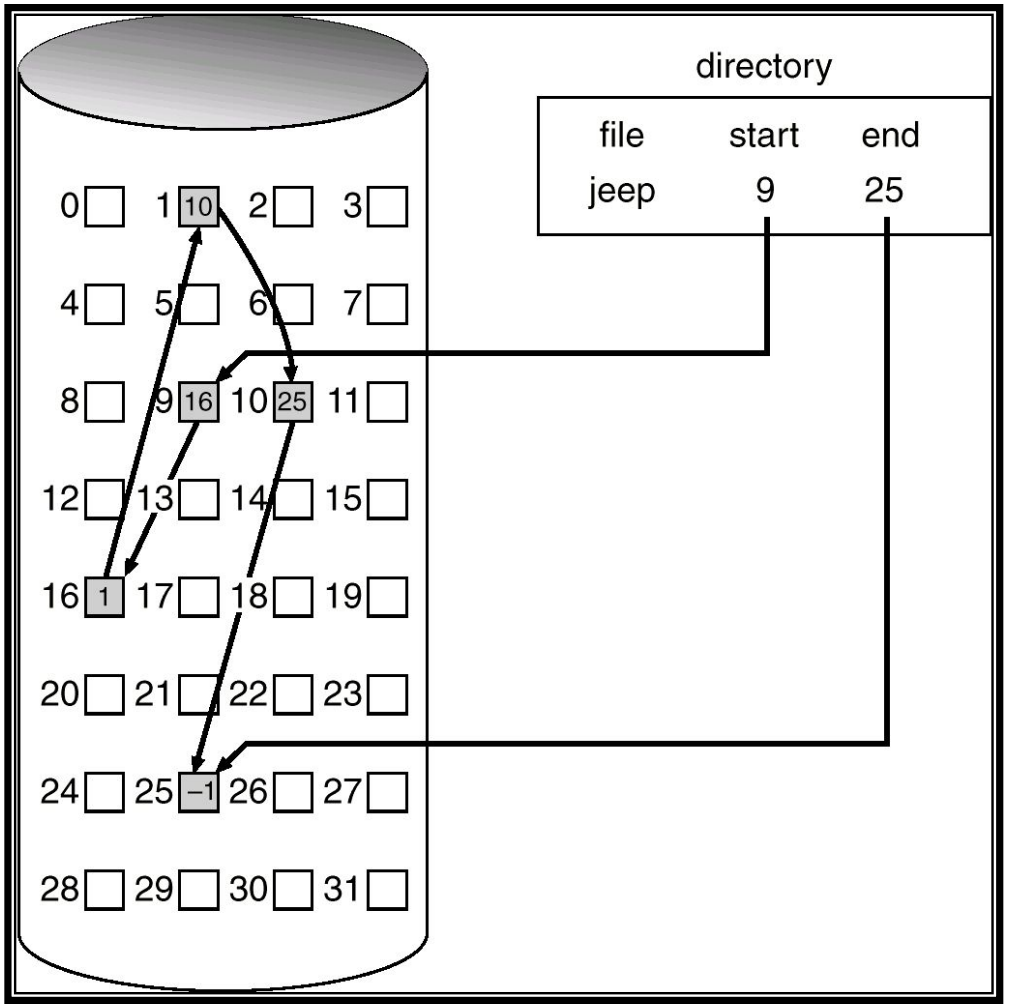
Схема организации виртуальной файловой системы



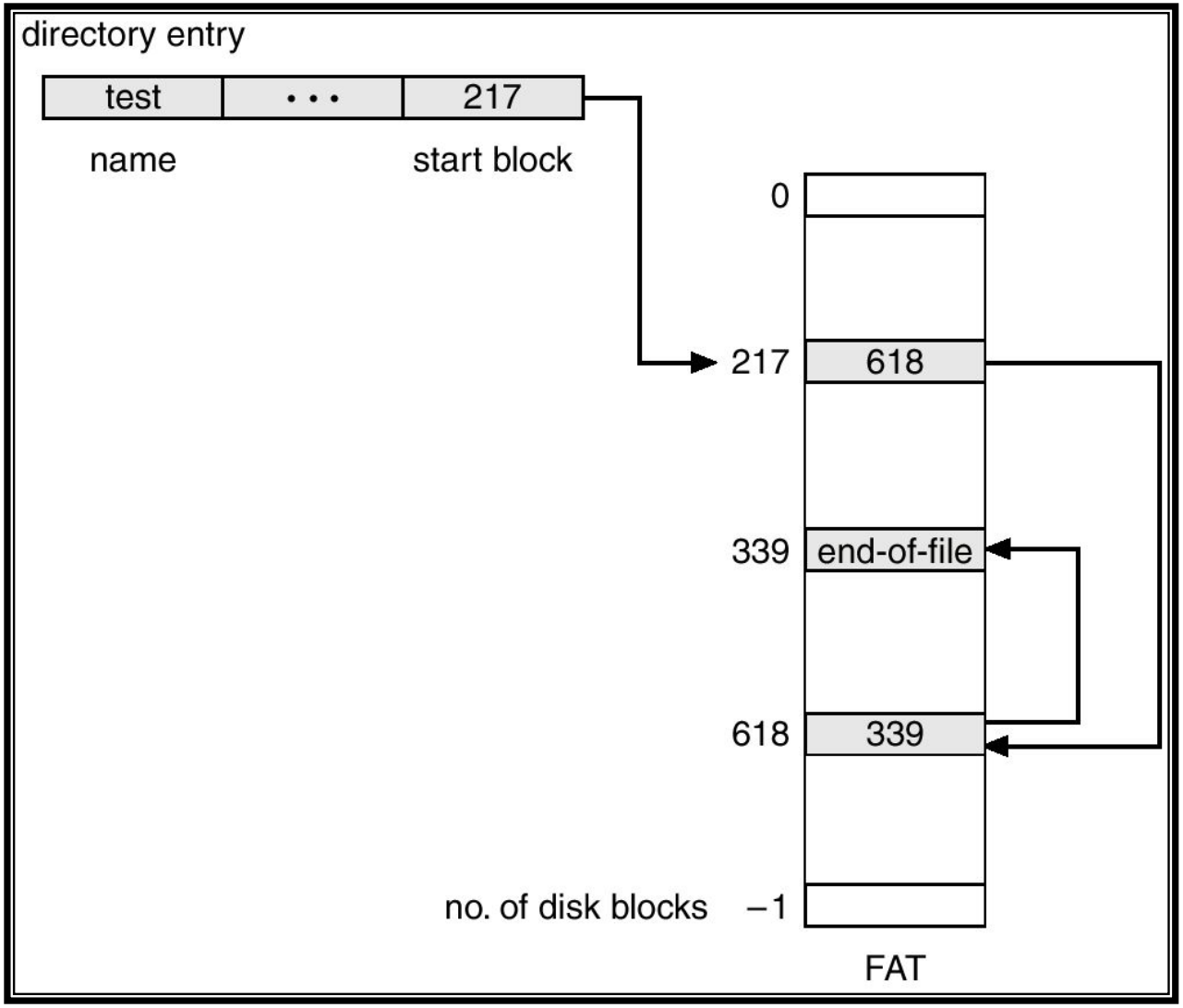
Смежное размещение файла на диске



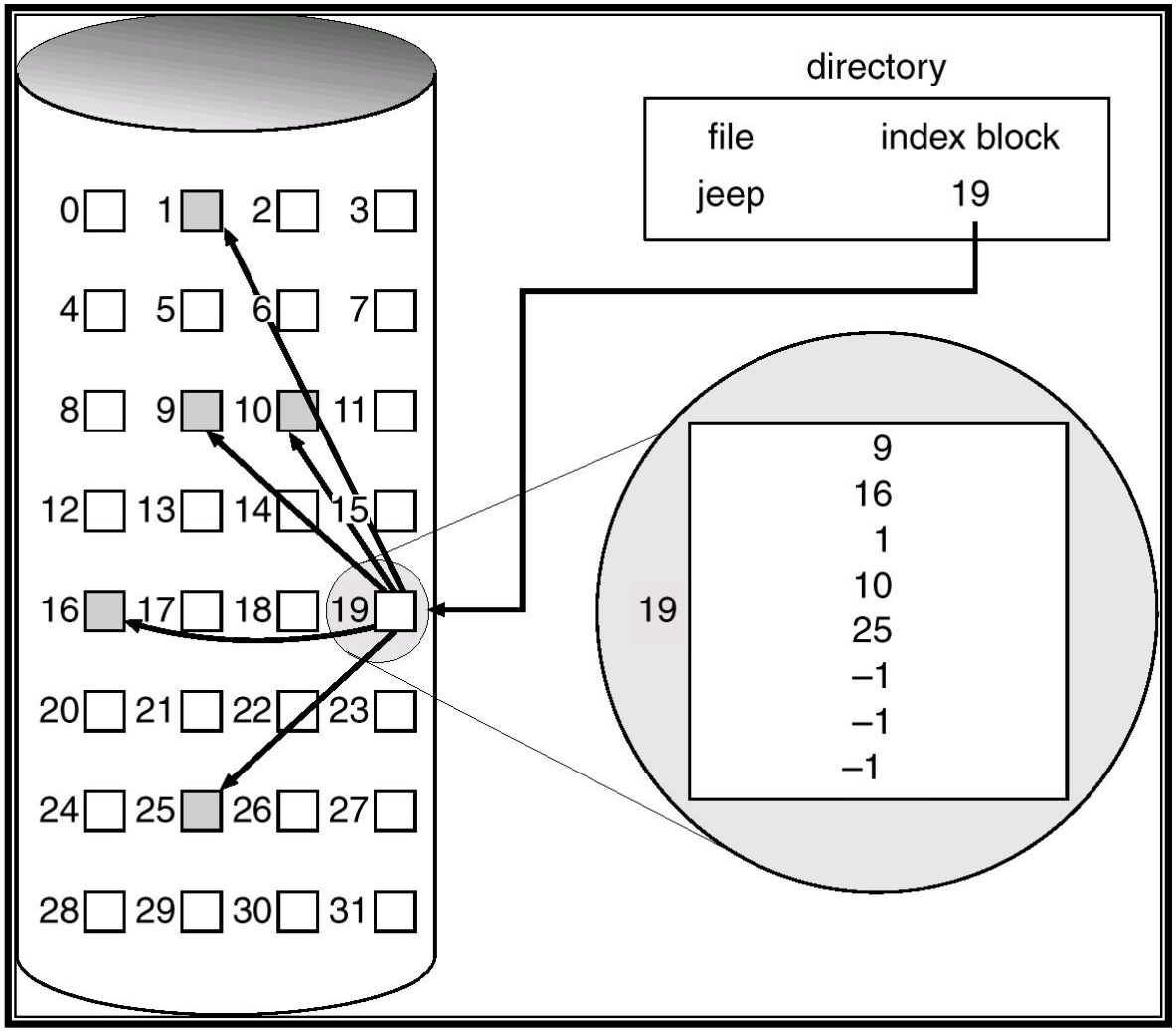
Ссылочное размещение файла



File Allocation Table (FAT)

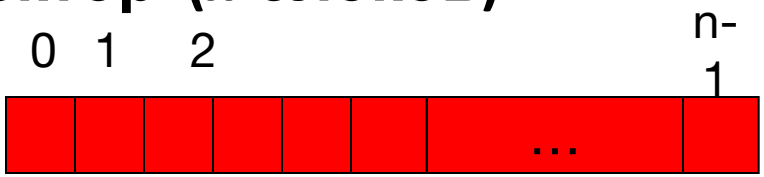


Пример индексируемого размещения



Управление свободной памятью

- **Битовый вектор (*n* блоков)**



bit[*i*] = 0 ⇒ block[*i*] свободен

1 ⇒ block[*i*] занят

Вычисление номера блока

(число битов в слове) *
(число нулевых слов) +
смещение первой 1

Управление свободной памятью (прод.)

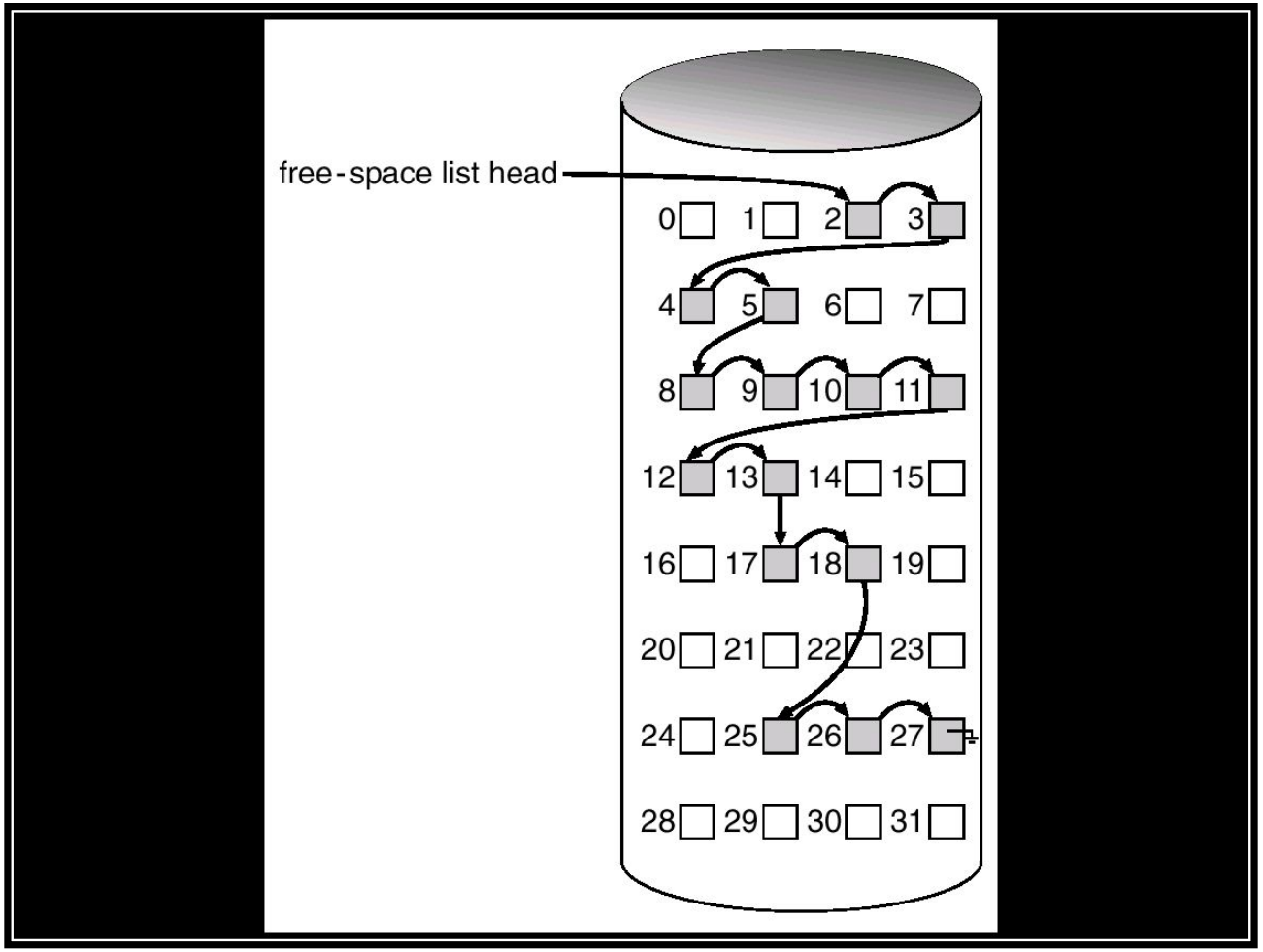
- **Битовые шкалы требуют дополнительной памяти. Пример:**

  размер блока = 2^{12} байтов

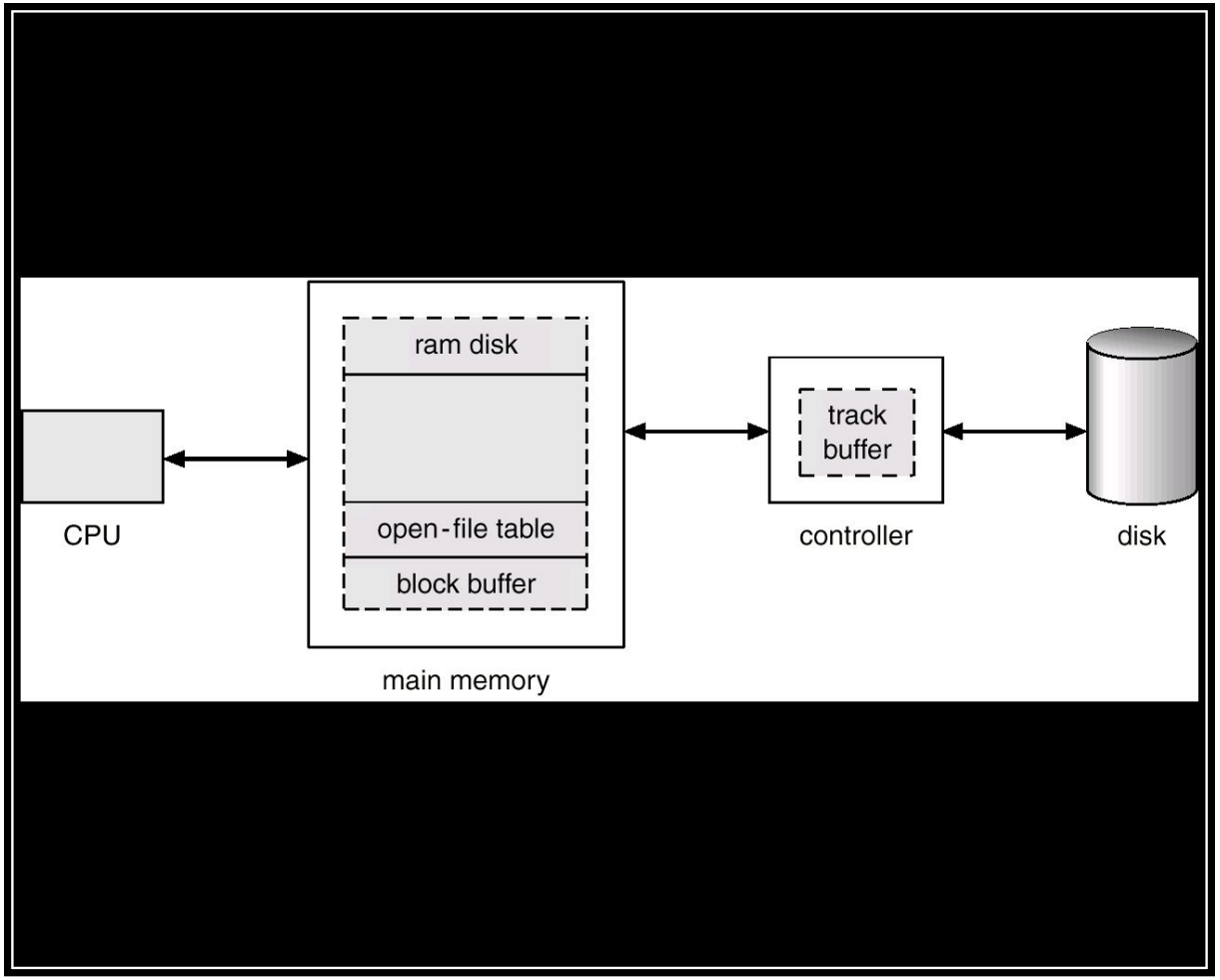
  размер диска = 2^{30} байтов (1 GB)

  n **= $2^{30}/2^{12} = 2^{18}$ битов (или 32 KB)**
- **Легко получать смежно расположенные файлы**
- **Связанный список (свободной памяти):**
  - **Невозможно легко получить смежную область памяти**
  - **Нет лишнего расходования памяти**

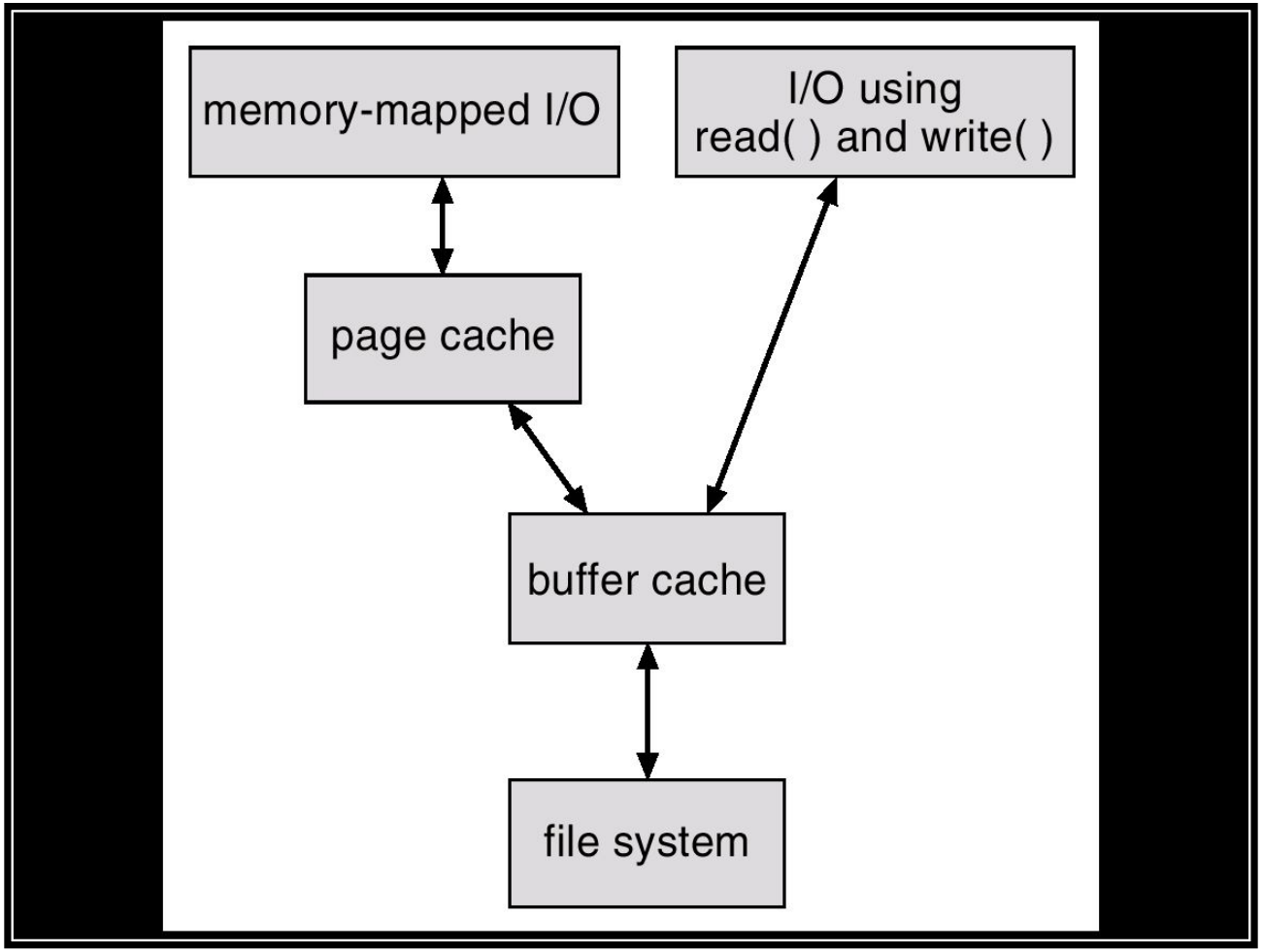
Связанный список свободной дисковой памяти



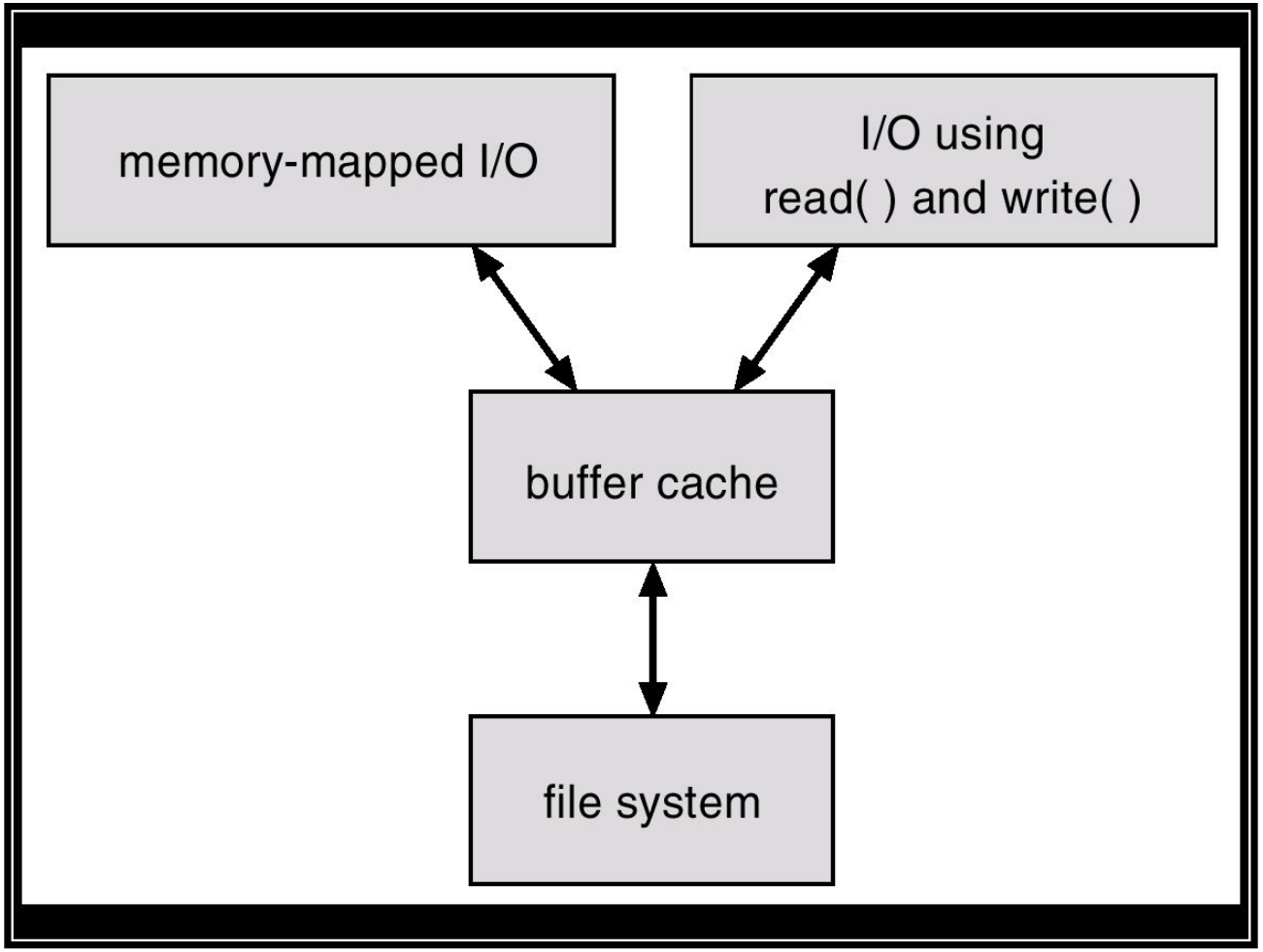
Различные методы размещения кэша для диска



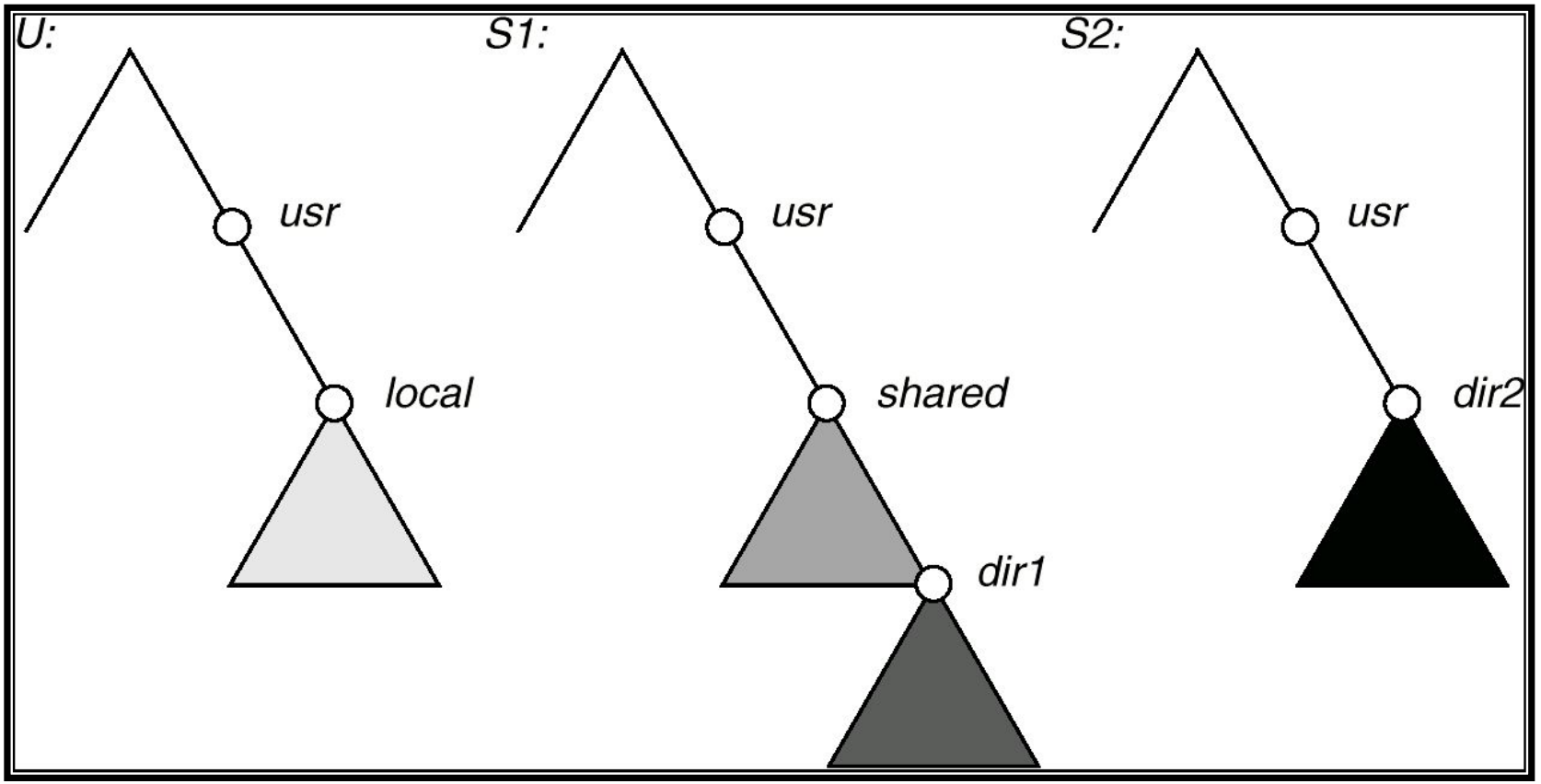
Ввод-вывод без унифицированной буферной кэш-памяти



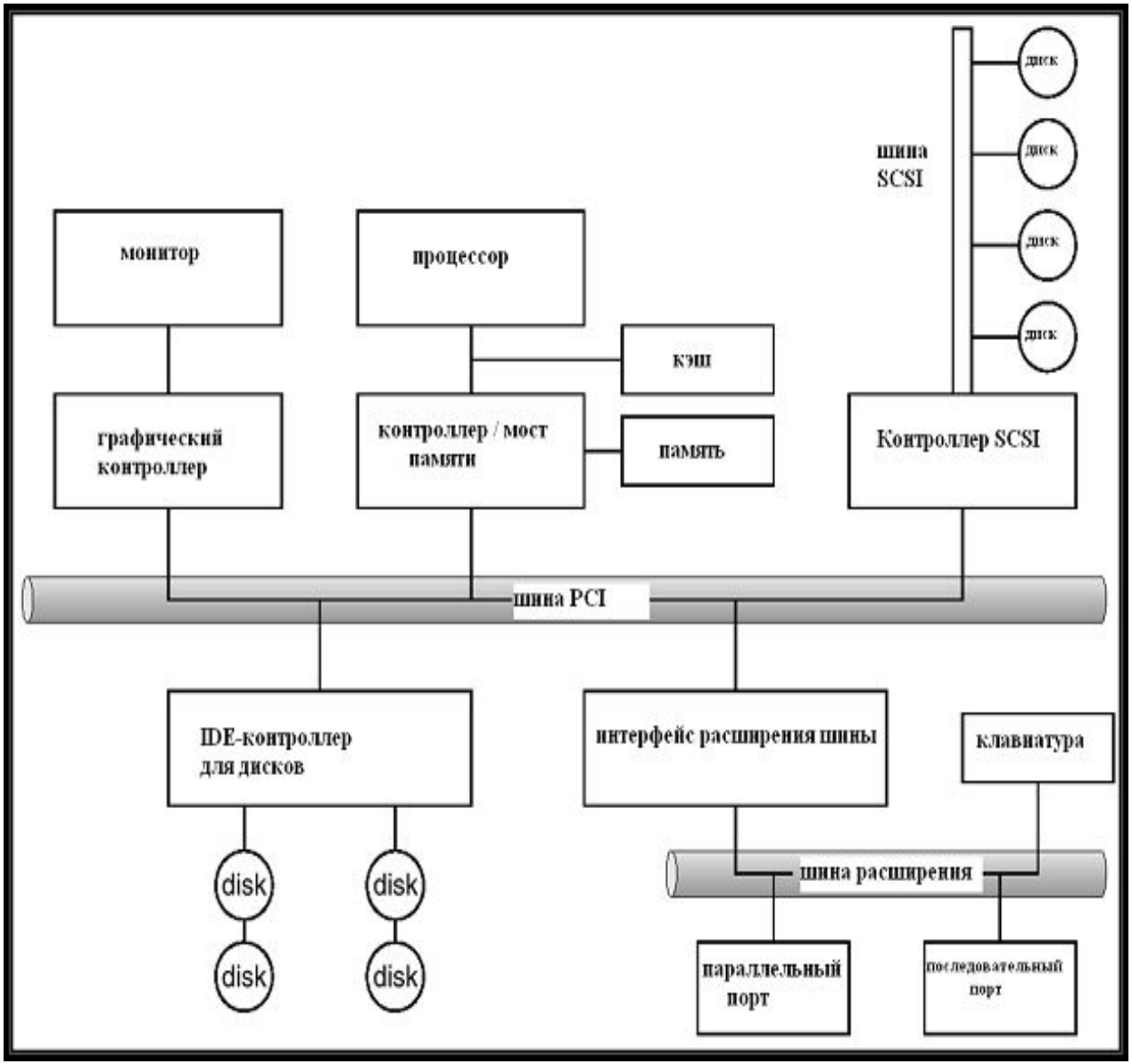
Ввод-вывод с использованием унифицированной буферной кэш-памяти



Три независимых файловых системы



Типовая структура шины ПК



IDE – типовой интерфейс для подключения внутри корпуса компьютера через **шлейфы** внутренних жестких дисков, устройств CD – и DVD-ROM.
В современных компьютерах для внутренних дисков вместо IDE используется более высокоскоростной интерфейс SATA.
Контроллер и шина SCSI – возможность подключения к одному SCSI-порту **цепочки (гирлянды) SCSI-устройств**(дисков, сканеров, устройств CD-ROM и DVD-ROM и др.), каждое из которых имеет свой, уникальный в данной цепочке, номер – **SCSI ID** от 0 до 9.

Расположение портов для устройств на ПК (частично)

диапазон адресов устройств ввода-вывода (шестнадцатиричных	устройство
000-00F	DMA-контроллер
020-021	контроллер прерываний
040-043	таймер
200-20F	игровой контроллер
2F8-2FF	последовательный порт (вторичный)
320-32F	контроллер жесткого диска
378-37F	параллельный порт
3D0-3DF	графический контроллер
3F0-3F7	контроллер гибких дисков (дискет)
3F8-3FF	последовательный порт (первичный)

Опрос устройств (polling)

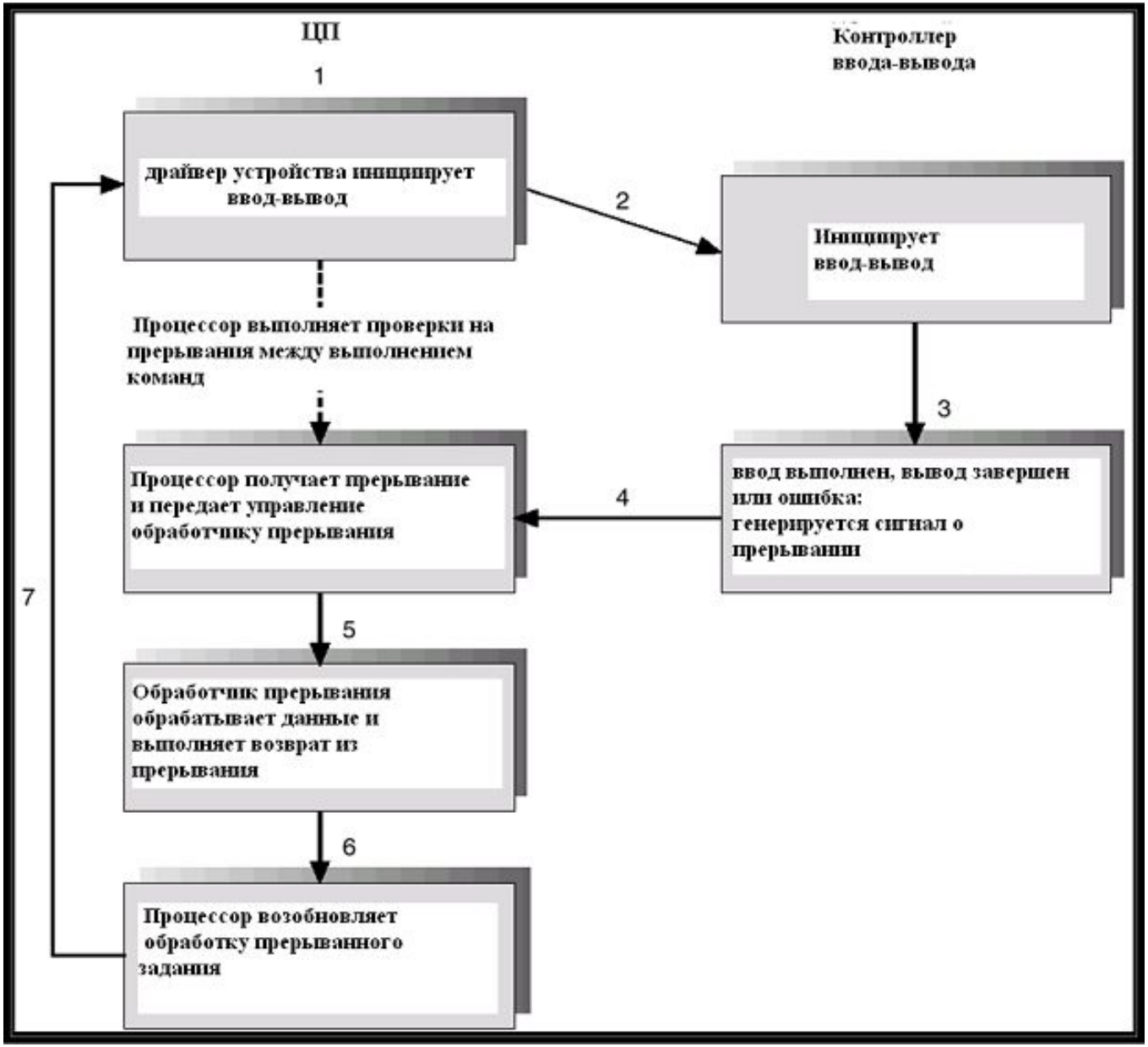
- **Определяет состояние устройства**
- command-ready – готово к выполнению команд;
- busy – занято;
- error – ошибка.
- **Цикл busy-wait ожидания ввода-вывода с устройством**

Прерывания

Линия **запросов на прерывания (interrupt request – IRQ)** переключается устройством ввода-вывода, которое сигнализирует с помощью запроса на прерывание о начале или окончании ввода-вывода.

Обработчик прерываний получает сигнал о прерывании. Сигнал может быть**замаскирован (maskable),** чтобы игнорировать или задержать прерывание – например, если прерывание произошло в обработчике другого прерывания.

Цикл ввода-вывода, управляемого прерываниями

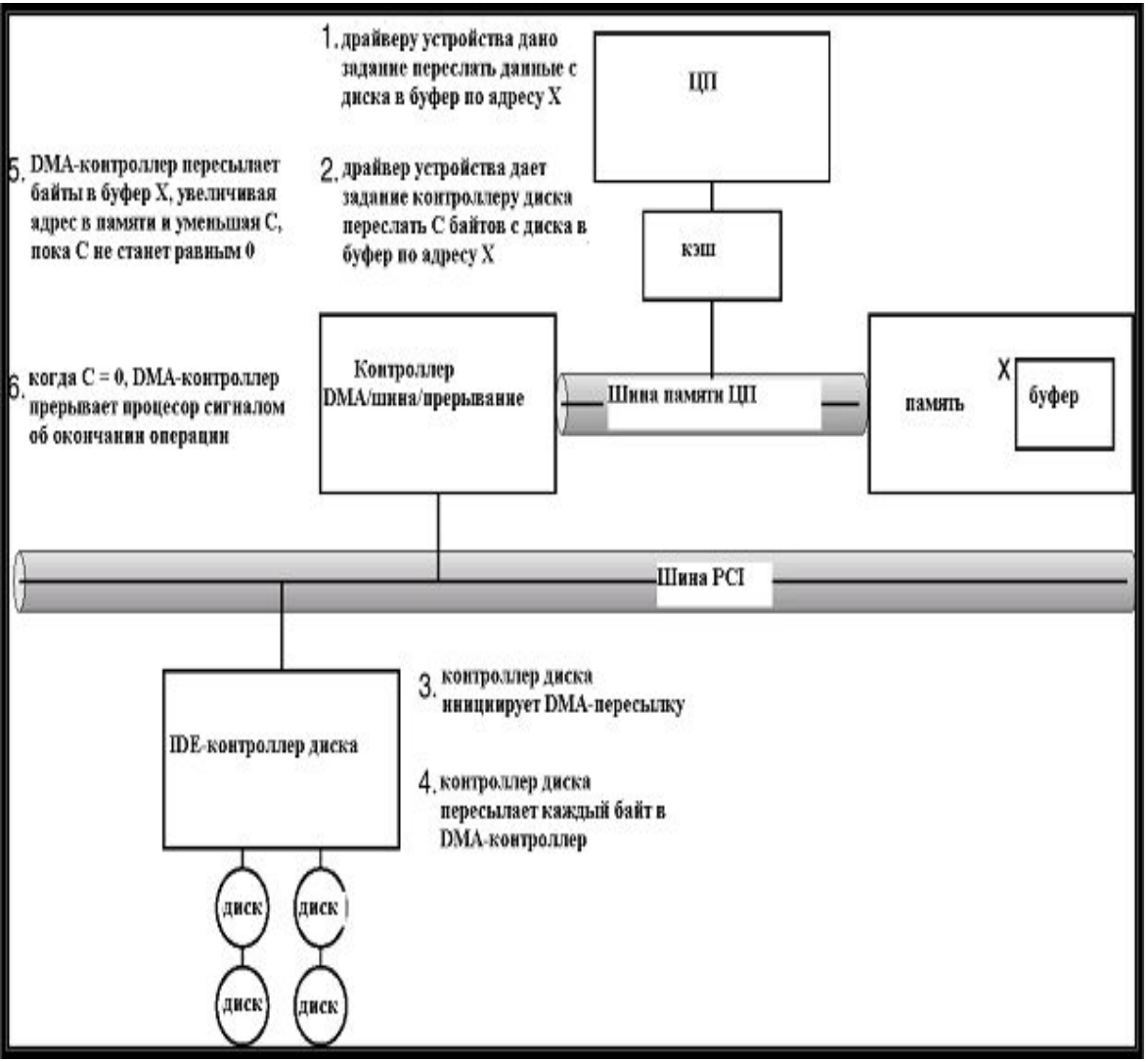


Вектор прерываний (событий) в процессоре Intel Pentium

Номер прерывания	Описание
0	ошибка при делении
1	исключение при отладке
2	прерывание по null
3	точка остановки
4	прерывание, обнаруженное INTO
5	исключение по выходу за границы
6	неверный код операции
7	устройство недоступно
8	двойное прерывание
9	переполнение сегмента сопроцессора
10	неверный сегмент состояния задачи
11	сегмент отсутствует
12	ошибка стека
13	общее прерывание по защите
14	отказ страницы
15	(зарезервировано Intel, не использовать)
16	ошибка в операции с плавающей точкой
17	контроль выравнивания
18	контроль аппаратуры
19-31	зарезервировано Intel, не использовать
32-255	маскируемые прерывания

Вектор прерываний – резидентный массив, содержащий адреса обработчиков прерываний в операционной системе, - используется с целью переадресовки прерывания для обработки соответствующим **обработчиком (handler).**Работа с вектором прерываний основана на приоритетах внешних устройств, инициировавших прерывания.

Процесс выполнения DMA (Direct Memory Access)



при трад организации i/o контроллер устр-ва исп собст буф память, что приводит к необходимости двойной пересылки данных – сначала процессор пересылает данные в буфер, созданный ОС, затем ОС пересылает данные в буфер устройства. **В-выв с прямым доступом к памяти (Direct Memory Access)** - более эффективная схема организации ввода-вывода, основанная на исп фрагмента основной памяти в качестве буфера устройства для выполнения ввода-вывода.

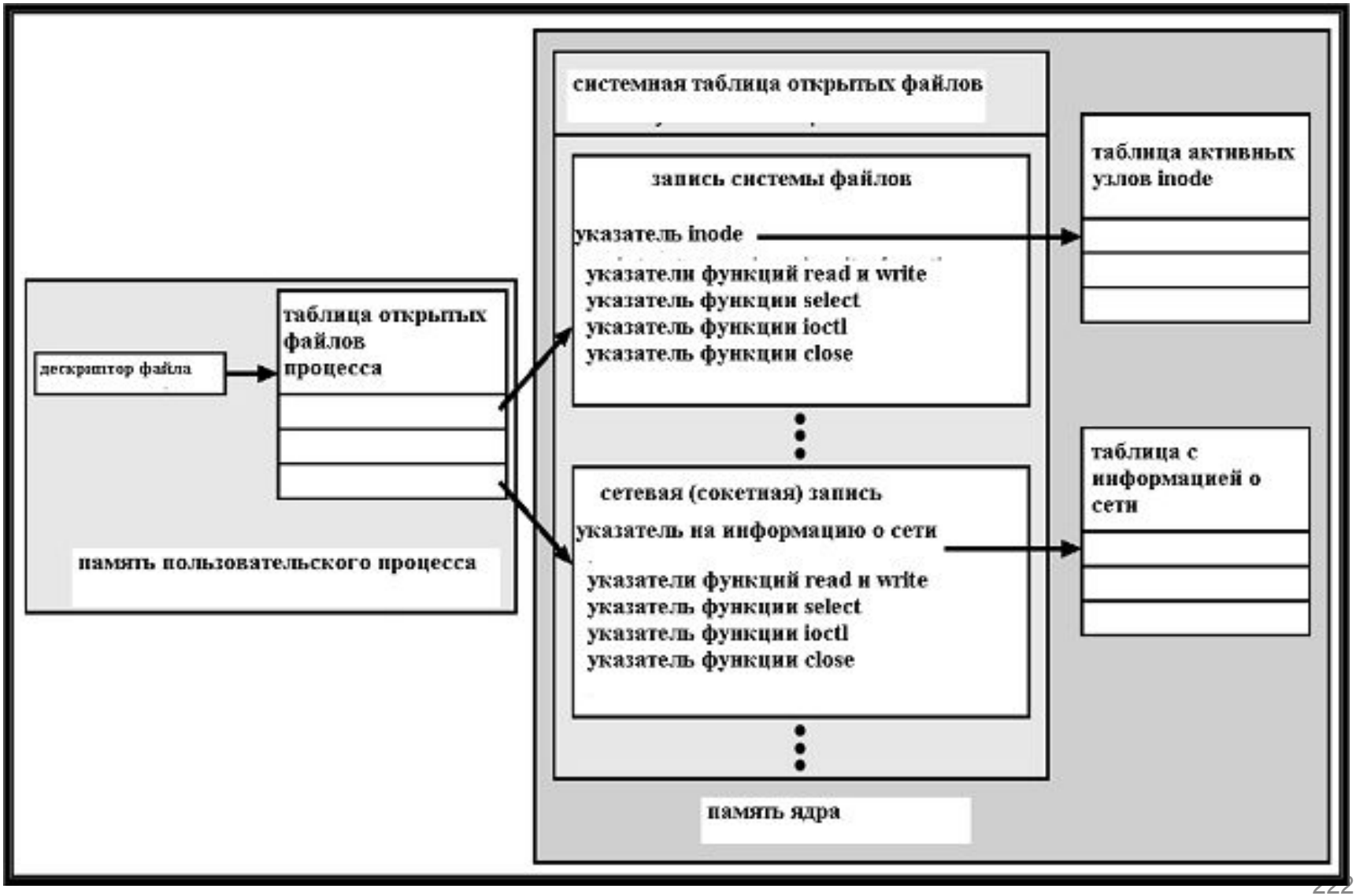
Структура модулей ввода-вывода в ядре

Более низкий уровень, уровень драйверов устройств, скрывает различия между контроллерами ввода-вывода конкретных устройств от ядра ОС.
Устройства вв-выврaзлич по многим параметрам в силу их специфики, например:
-Устройство для работы с потоками символов или с блоками;
-Устройство последовательного или прямого доступа;
-Разделяемое или специализированное (монополизируемое) устройство;
-Различия по скорости вып операций устройствами;
-Устройство для чтения/записи, или только для чтении, или только для записи.

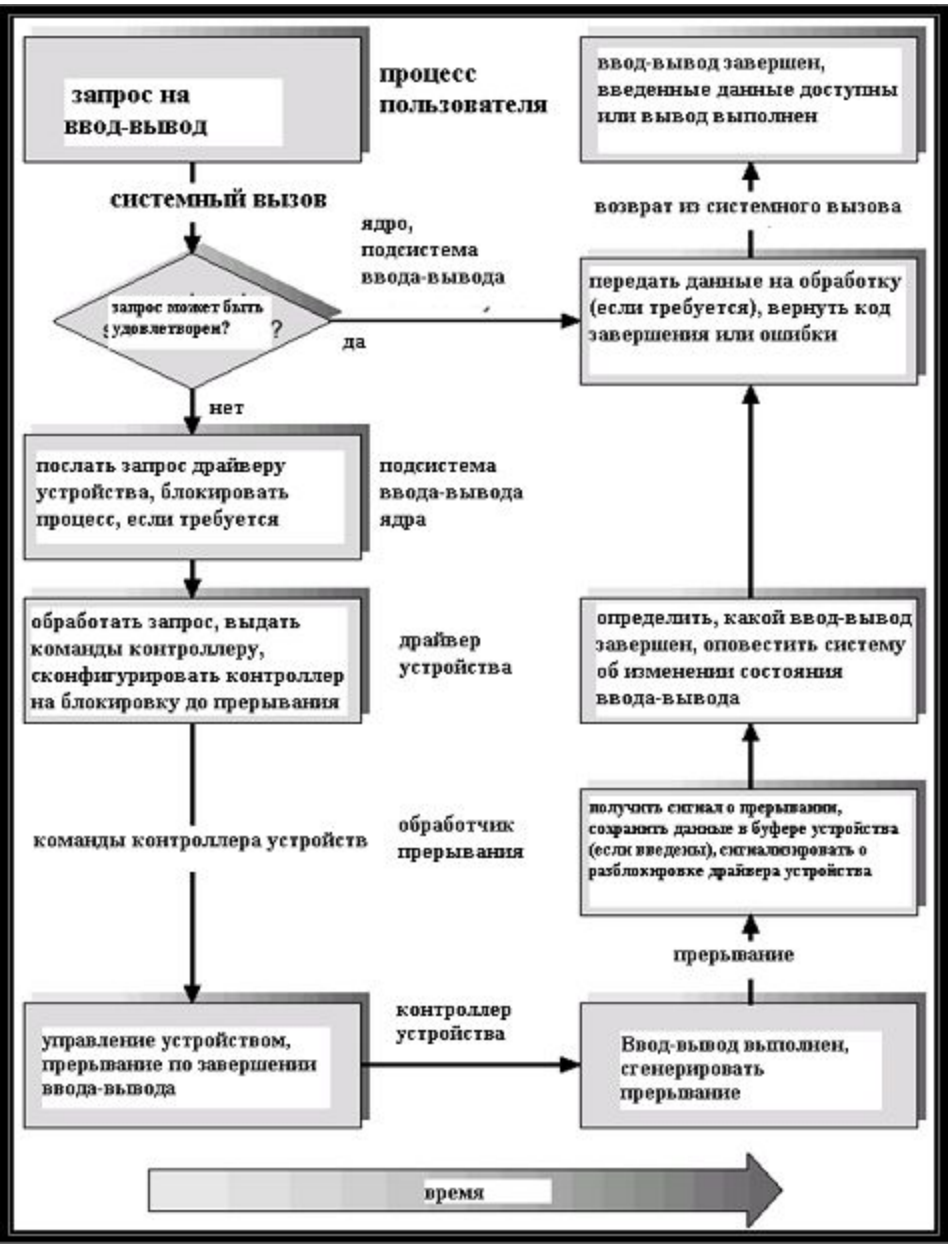
Характеристики устройств ввода-вывода

классификация	**варианты**	**примеры**
режим передачи данных	•символьный •блочный	•терминал •диск
метод доступа	•последовательный •произвольный	•модем •CD-ROM
метод передачи данных	•синхронный •асинхронный	•лента •клавиатура
возможность совместного доступа	•монопольный •общий	•лента •клавиатура
скорость устройства	•латентность •время поиска •скорость передачи •задержка между операциями	
направленнность ввода-вывода	•только чтение •только запись •чтение-запись	•CD-ROM •графический контроллер •диск

Структура модулей ввода-вывода в ядре UNIX

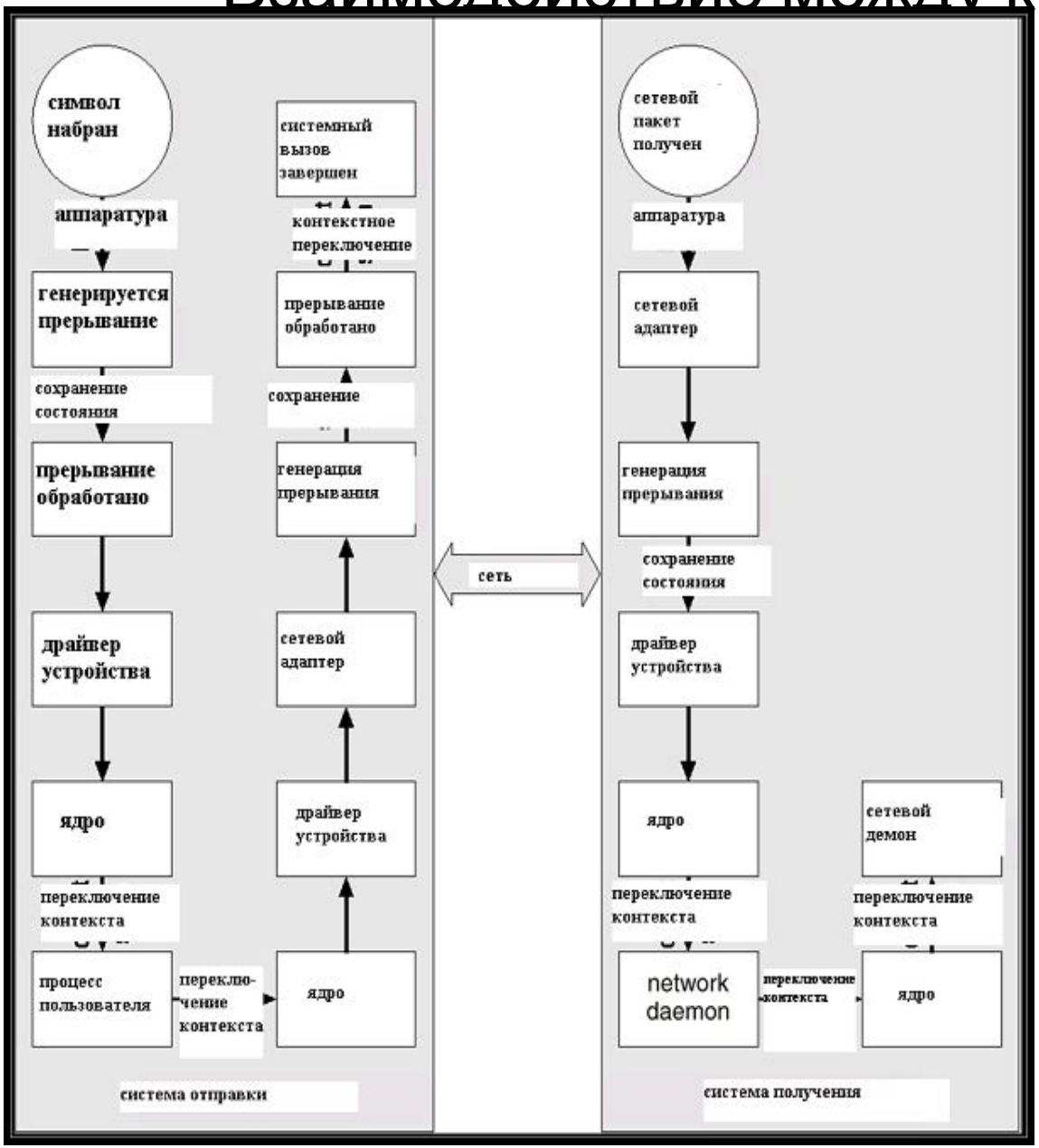


Жизненный цикл запроса на ввод-вывод



процесс чтения из дискового файла состоит из следующих этапов:
-Определяется устройство, на котором хранится файл;
-Выполняется трансляция имени в представление устройства;
- Физически считанные данные с диска размещаются в буфере;
-Данные становятся доступными для запросившего их процесса;
-Управление возвращается процессу.
I/O – важный фактор в производит-сти системы. Имеются несколько факторов, опред, насколько i-o критичен по эффективности в системе:-i/o требует от проца исполнения драйвера устройства - кода уровня ядра ОС;
-Необходимо вып контекстные переключ, связанные с прерываниями;
-Необходимо вып копир данных

Взаимодействие между компьютерами



Для повышения производительности ввода-вывода и сетевого взаимодействия в системе необходимо:

-Сократить число контекстных переключений;

-Сократить объем копирования данных;

-Сократить число прерываний, используя большие переходы, интеллектуальные контроллеры и опрос устройств;

Использовать DMA (Direct Memory Access);

-Сбалансировать нагрузку на процессор, память и шину и производительность ввода-вывода с целью повышения суммарной производительности.

.

Структура STREAMS