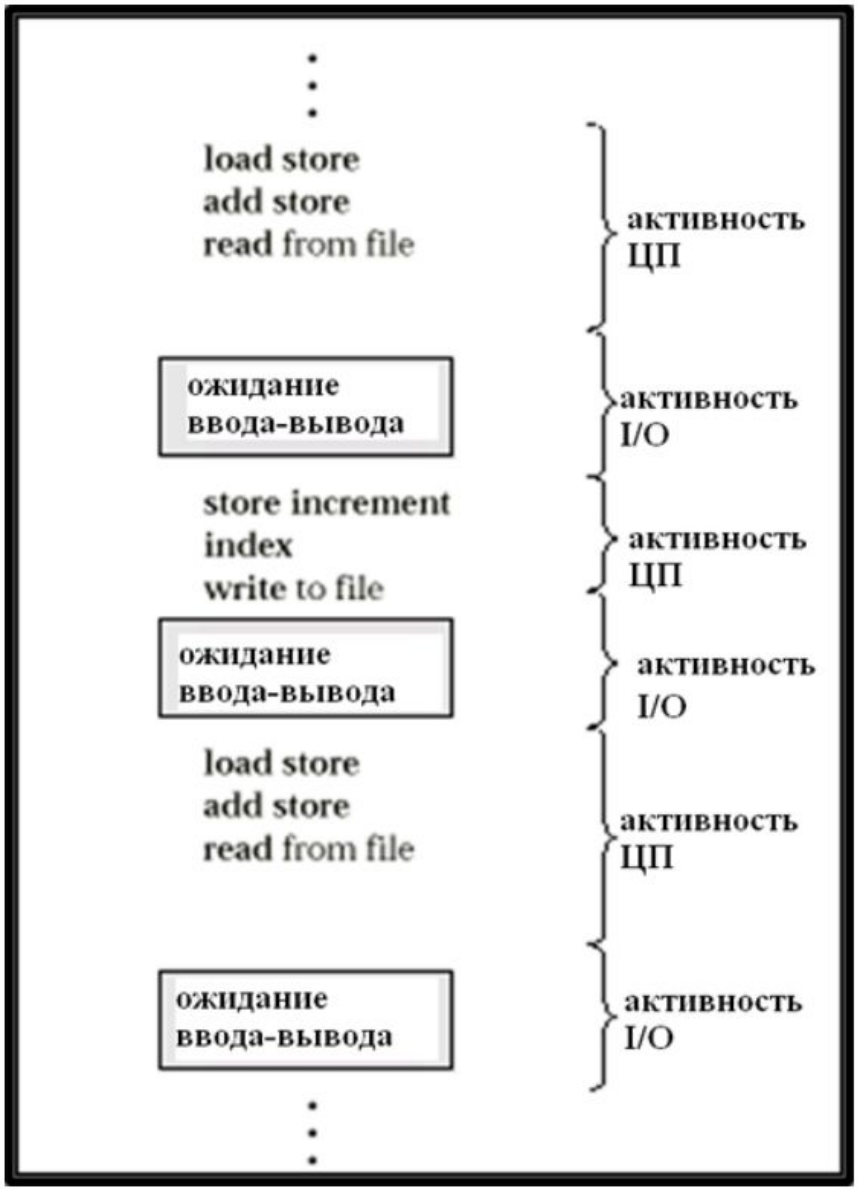
Операционные системы

Стратегии и критерии диспетчеризации процессов.

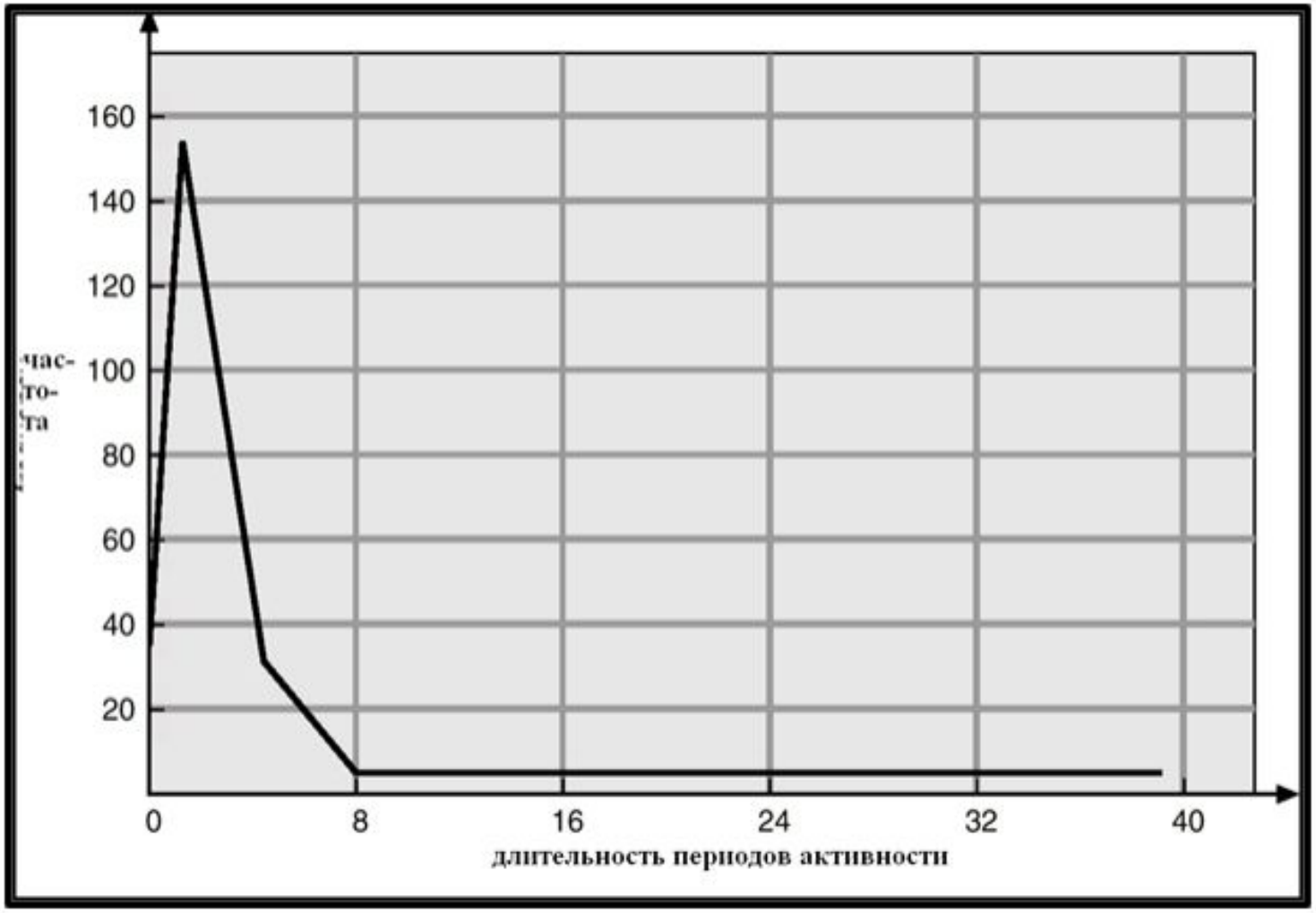
Основные понятия диспетчеризации процессора

- **Цель – максимальная загрузка процессора. Достигается п помощью мультипрограммирования**
- **Цикл CPU / I-O – Исполнение процесса состоит из работы процессора и ожидания ввода-вывода.**
- **Распределение периодов активности процессора**

Последовательность активных фаз (bursts) процессора и ввода-вывода



Гистограмма периодов активности процессора



Планировщик процессора (scheduler)

- **Выбирает один из нескольких процессов, загруженных в память и готовых к выполнению, и выделяет процессор для одного из них.**
- **Решения по диспетчеризации могут быть приняты, когда процесс:**
  1. **Переключается из состояния выполнения в состояние ожидания.**
  2. **Переключается из состояния выполнения в состояние готовности к выполнению.**
  3. **Переключается из состояния ожидания в состояние готовности.**
  4. **Завершается.**
- **Диспетчеризация типов 1 и 4 – *не опережающая (non-preemptive)*.**
- **В остальных случаях – *опережающая (preemptive)*.**

диспетчер

- **Модуль диспетчера предоставляет процессор тому процессу, который был выбран планировщиком, то есть:**
  - **Переключает контекст**
  - **Переключает процессор в пользовательский режим**
  - **Выполняет переход по соответствующему адресу в пользовательскую программу для ее рестарта**
- ***Dispatch latency* – время, требуемое для диспетчера, чтобы остановить один процесс и стартовать другой.**

Критерии диспетчеризации

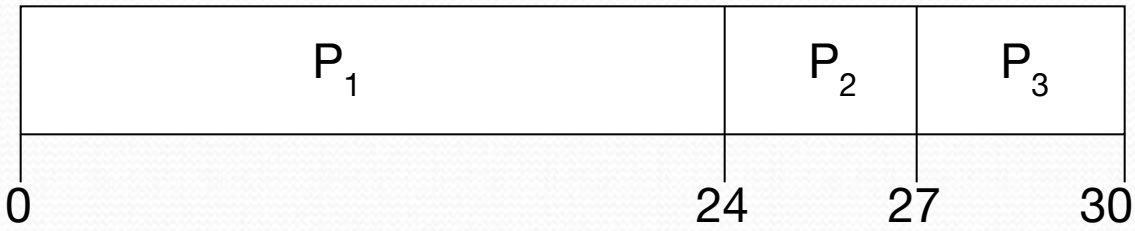
- **Использование процессора – поддержание его в режиме занятости, насколько это возможно**
- **Пропускная способность (throughput) – число процессов, завершающих свое выполнение за единицу времени**
- **Время обработки (turnaround time) – время, необходимое для исполнения какого-либо процесса**
- **Время ожидания (waiting time)– время, которое процесс ждет в очереди процессов, готовых к выполнению**
- **Время ответа (response time) – время, требуемое от момента первого запроса до первого ответа (для среды разделения времени)**

Стратегия диспетчеризации «обслуживание в порядке поступления» First-Come-First-Served (FCFS)

ресурсы процессора предоставляются процессам в порядке их поступления (ввода) в систему, независимо от потребляемых ими ресурсов

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:



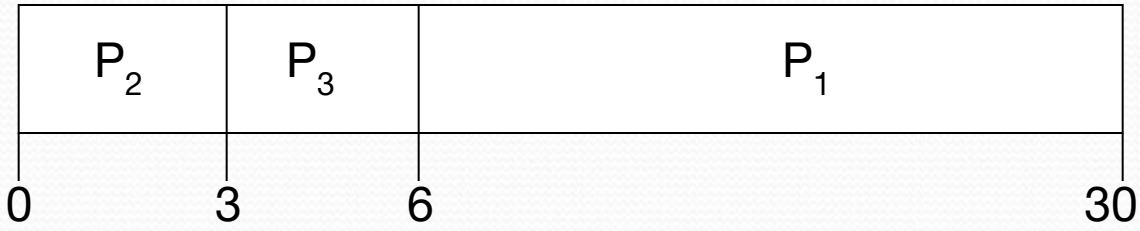
- Время ожидания для P_1 = 0; P_2 = 24; P_3 = 27
- Среднее время ожидания: (0 + 24 + 27)/3 = 17

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

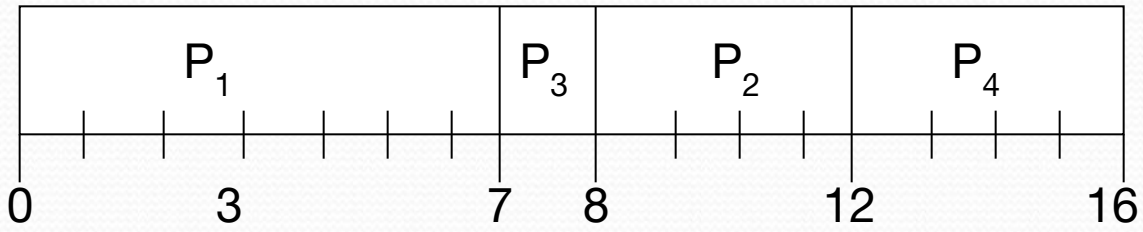
Стратегия Shortest-Job-First (SJF) обслуживание самого короткого задания первым

- **С каждым процессом связывается длина его очередного периода активности. Эта длина используется для того, чтобы первым обслужить самый короткий процесс .**
- **Две схемы:**
  - **Без опережения– пока процессу предоставляется процесс, он не может быть прерван, пока не истечет его кварнт времени.**
  - **С опережением – если приходит новый процесс, время активности которого меньше, чем оставшееся время активного процесса, - прервать активный процесс. Эта схема известна под названием Shortest-Remaining-Time-First (SRTF).**
- **SJF оптимальна – обеспечивает минимальное среднее время ожидания для заданного набора процессов.**

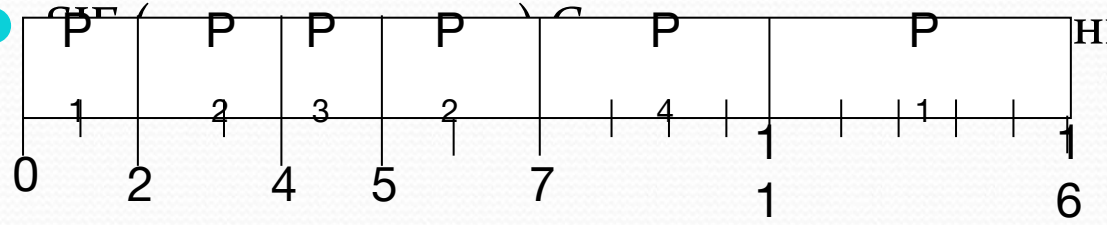
Пример:

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 (без опережения) Среднее время ожидания (0 + 6 + 3 + 7)/4 = 4



- SJF (P P P P) C P P ния (9 + 1 + 0 +2)/4 = 3



вследствие применения принципа прерывания процессов, периоды непрерывного

Определение длины следующего периода активности

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

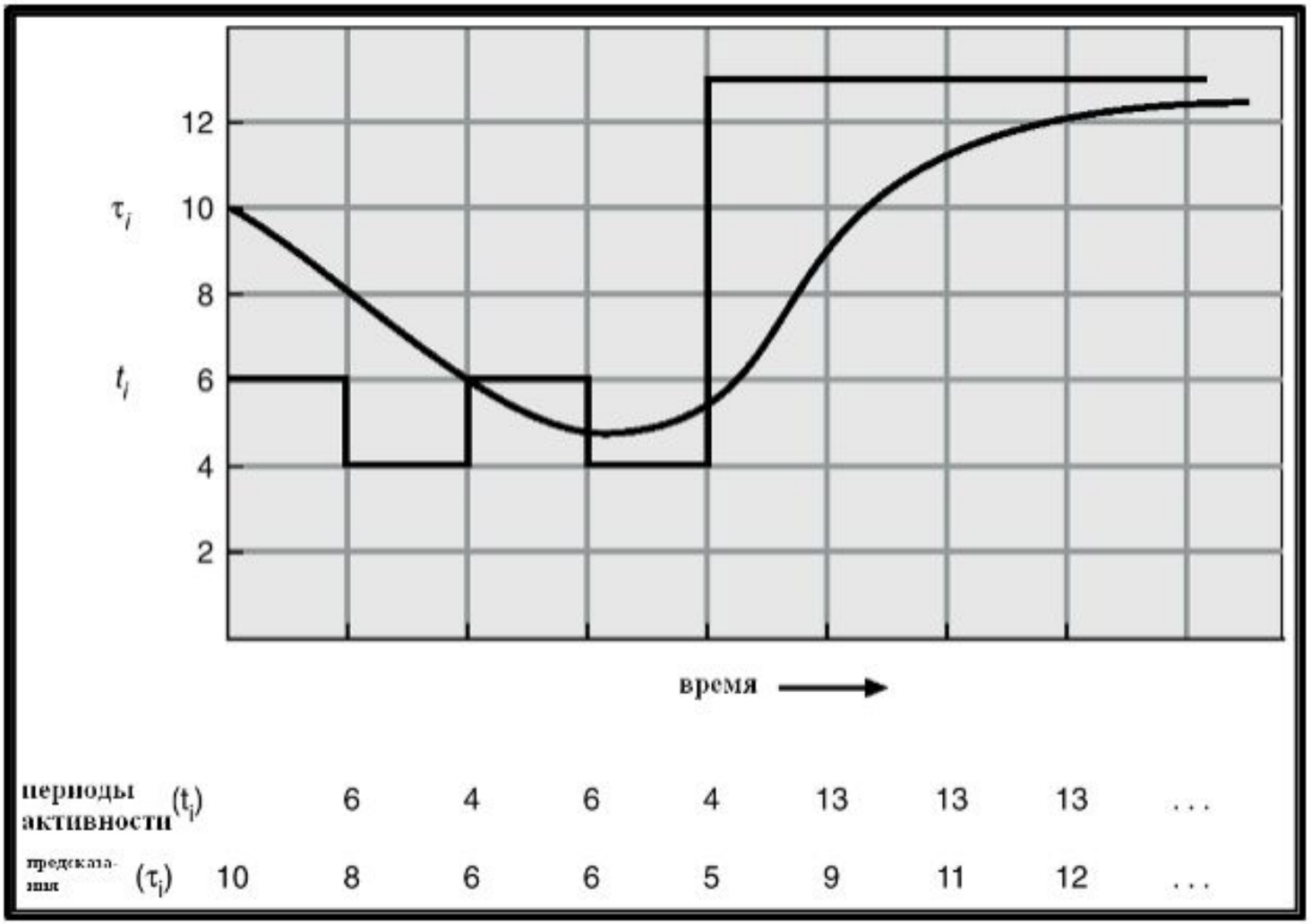
- t_n – фактическая длина n-го периода активности процесса;
- τ_n – предсказанная длина n-го периода активности процесса.

Будем искать значение τ_{n+1} для предсказания следующего периода активности процесса как следующую линейную комбинацию t_n и τ_n:

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

где α – число между 0 и 1. Коэффициент α характеризует, в какой степени при предсказании учитывается недавняя история вычислений.

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:**

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

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

Диспетчеризация по приоритетам

- **С каждым процессом связывается его приоритет (целое число)**
- **Процессор выделяется процессу с наивысшим приоритетом (меньшее число – высший приоритет)**
- **Стратегии с опережением и без опережения**
- **SJF – это диспетчеризация по приоритетам, в которой приоритетом является очередное время активности.**
- **Проблема ≡ “Starvation” (“голодание”) – процессы с низким приоритетом могут никогда не исполниться**
- **Решение ≡ “Aging” (“возраст”) – с течением времени приоритет процесса повышается.**

Стратегия Round Robin (RR)–круговая система

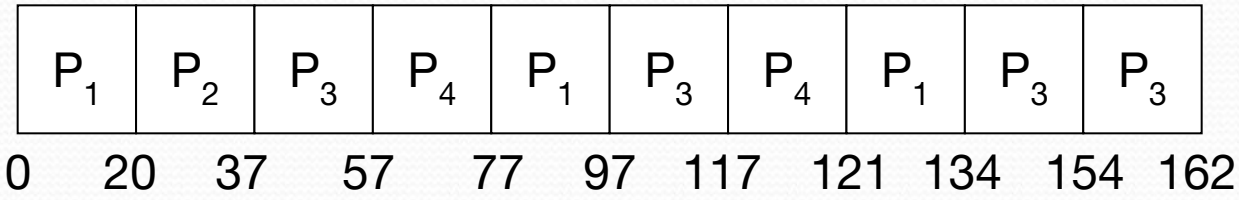
- **Каждый процесс получает небольшой квант процессорного времени, обычно – 10-100 миллисекунд. После того, как это время закончено, процесс прерывается и помещается в конец очереди готовых процессов.**
- **Если всего *n* процессов в очереди готовых к выполнению, и квант времени - *q*, то каждый процесс получает 1/*n* процессорного времени порциями самое большее по *q* единиц за один раз. Ни один процесс не ждет больше, чем (*n*-1)*q* единиц времени.**
- **Производительность**
  - ***q* велико ⇒ FIFO**
  - ***q* мало ⇒ *q* должно быть большим, по сравнению со временем контекстного переключения, иначе слишком велики накладные расходы**

Пример RR (квант времени = 20)

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

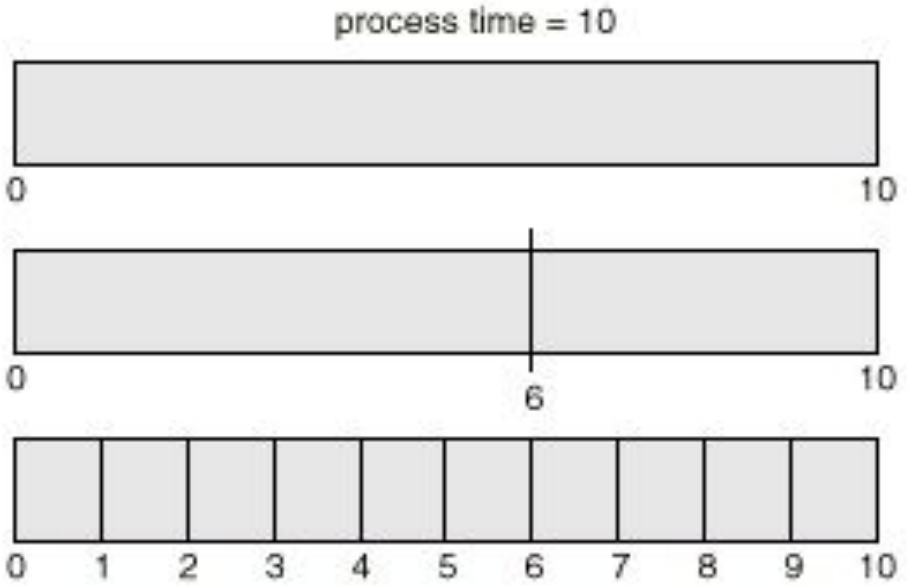
Процес	**Время активности**
P_1	53
P_2	17
P_3	68
P_4	24

- **Диаграмма Ганта:**



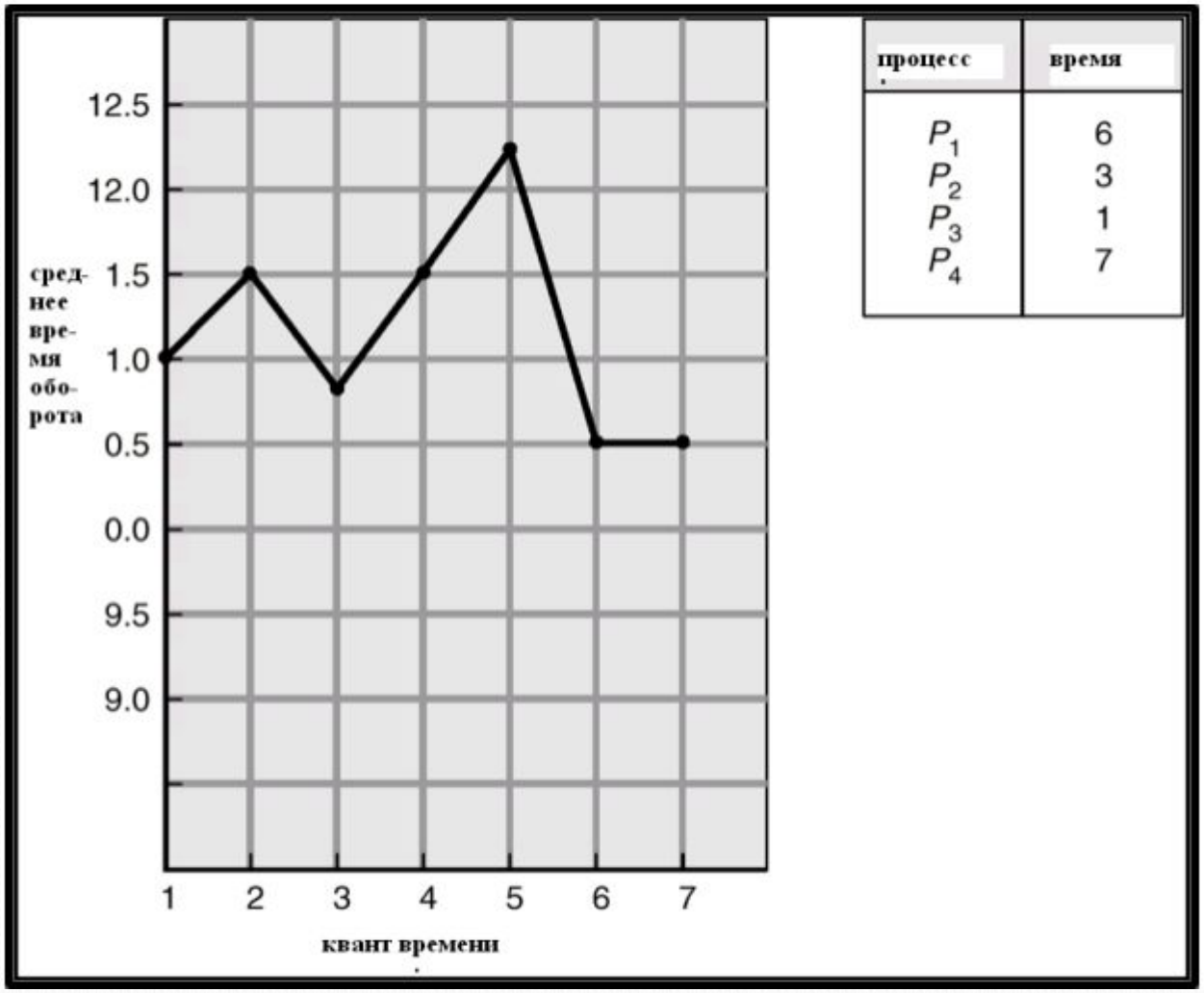
- **Обычно RR имеет худшее время оборота, чем SJF, но лучшее время ответа.**

Квант времени ЦП и время переключения контекста



quantum	context switches
12	0
6	1
1	9

Изменение времени оборота, в зависимости от кванта времени

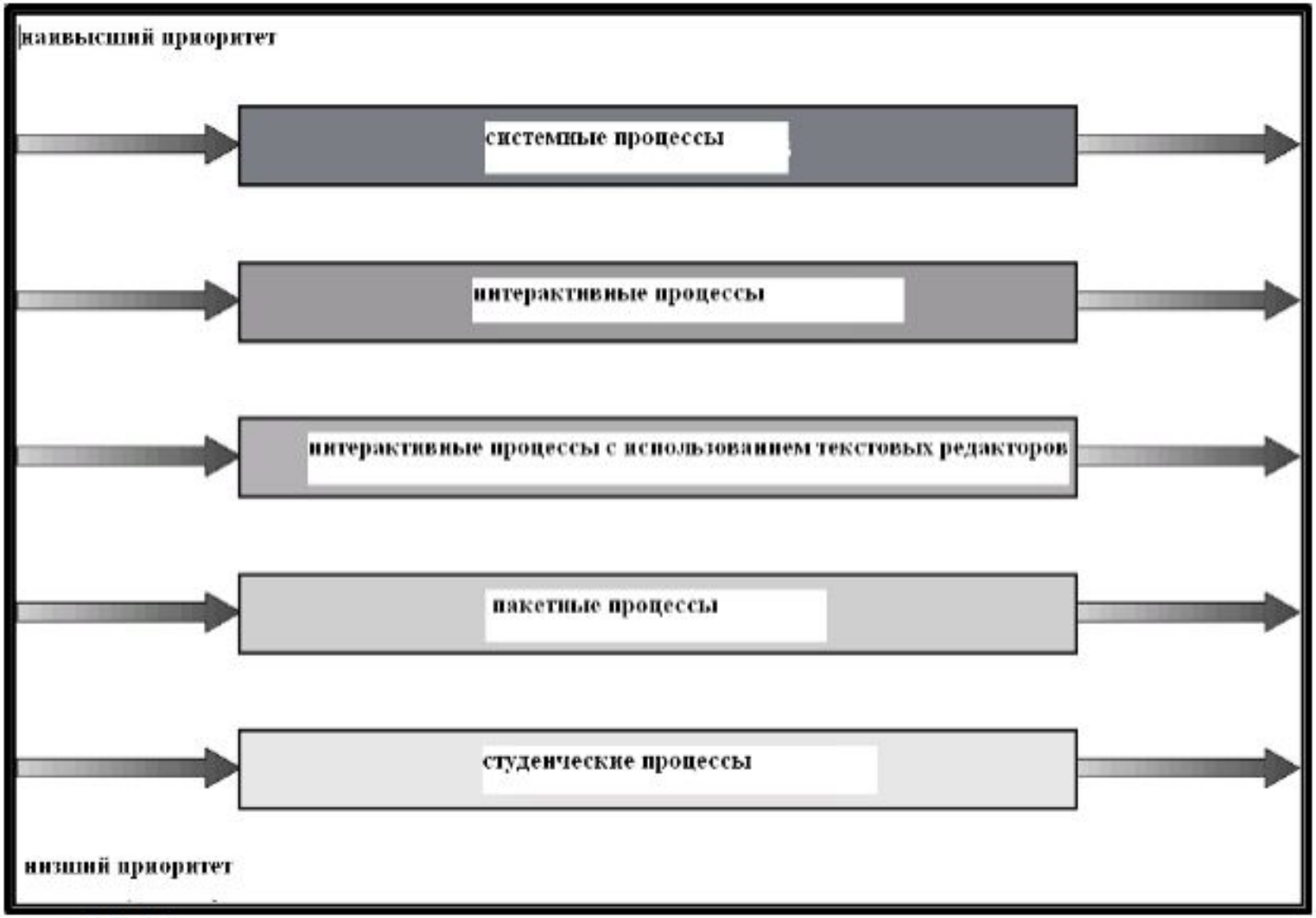


процесс	время
P_1	6
P_2	3
P_3	1
P_4	7

Многоуровневая очередь

- **Очередь готовых к выполнению процессов делится на две очереди:**
  основная (интерактивные процессы)
  фоновая (пакет)
- **Каждая очередь имеет свой собственный алгоритм диспетчеризации:**
  основная– RR
  фоновая – FCFS
- **Необходима также диспетчеризация между очередями.**
  - **С фиксированным приоритетом; (обслуживание всех процессов из основной очереди, затем – из фоновой). Есть вероятность “голодания”.**
  - **Выделение отрезка времени – каждая очередь получает некоторый отрезок времени ЦП, который она может распределять между процессами; например, 80 % - на RR в основной очереди; 20% на FCFS в фоновой очереди**

Диспетчеризация по принципу многоуровневой очереди



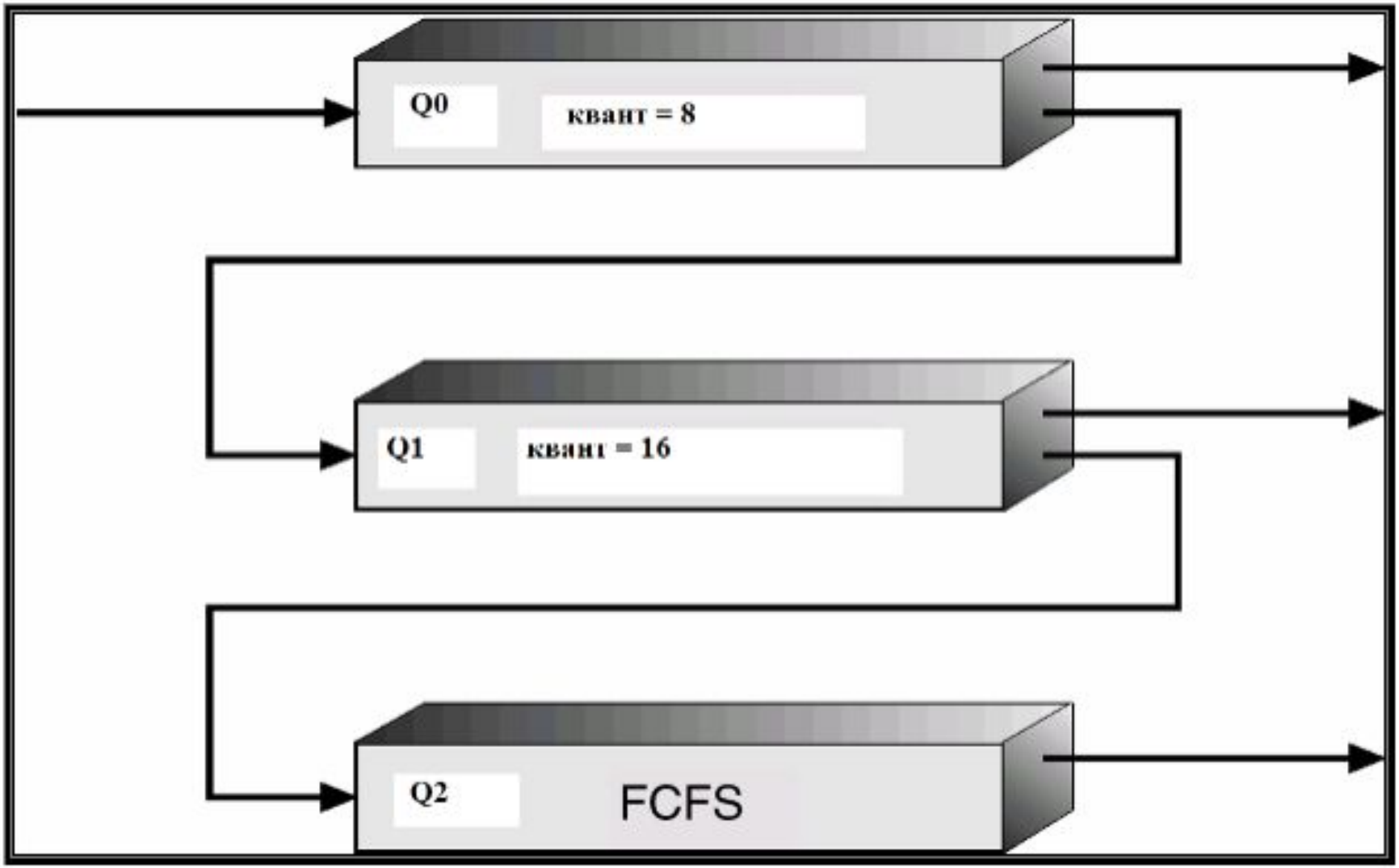
Многоуровневая аналитическая очередь (multi-level feedback queue)

- **Процесс может перемещаться из одной очереди в другую; увеличение его “возраста” (aging) может быть реализовано следующим образом:**
- **Планировщик многоуровневой аналитической очереди (multilevel-feedback-queue scheduler) определяется следующими параметрами:**
  - **Число очередей**
  - **Алгоритмы планирования для каждой очереди**
  - **Методы, используемые для определения, когда необходимо повысить уровень процесса**
  - **Методы, используемые для определения, когда необходимо понизить уровень процесса**
  - **Методы, используемые для определения, в какую именно очередь следует включить (новый) процесс, требующий обслуживания**

Пример многоуровневой аналитической очереди

- **Три очереди:**
  - **Q_0 – квант времени - 8 миллисекунд**
  - **Q_1 – квант времени – 16 миллисекунд**
  - **Q_2 – FCFS**
- **Планирование**
  - **Новое задание помещается в очередь Q_0, которая обслуживается по стратегии FCFS. Когда оно получает процессор, заданию дается 8 миллисекунд. Если оно не завершается за 8 миллисекунд, оно перемещается в очередь Q_1.**
  - **В очереди Q_1 задание вновь обслуживается по стратегии FCFS и получает 16 дополнительных миллисекунд. Если оно и после этого не завершается, оно прерывается и перемещается в очередь Q_2.**

Многоуровневые аналитические очереди



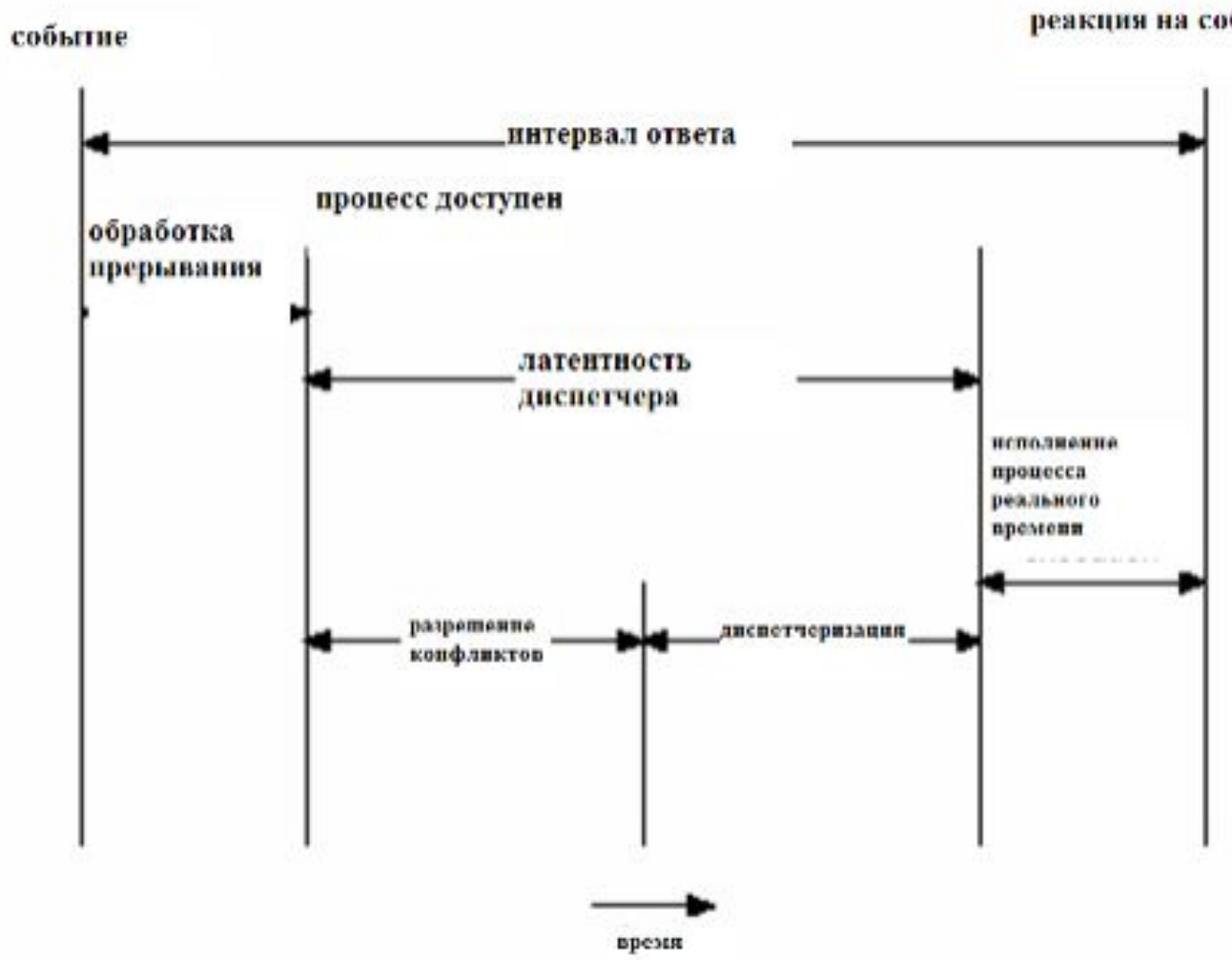
Планирование загрузки многопроцессорных систем

- **Планирование загрузки процессора более сложно, когда доступны несколько процессоров**
- ***Однородные процессоры* в многопроцессорной компьютерной системе**
- ***Параллельная загрузка (load sharing)***
- ***Асимметричное мультипроцессирование* – только одному процессору доступны системные структуры данных, что исключает необходимость в синхронизации по общим данным**

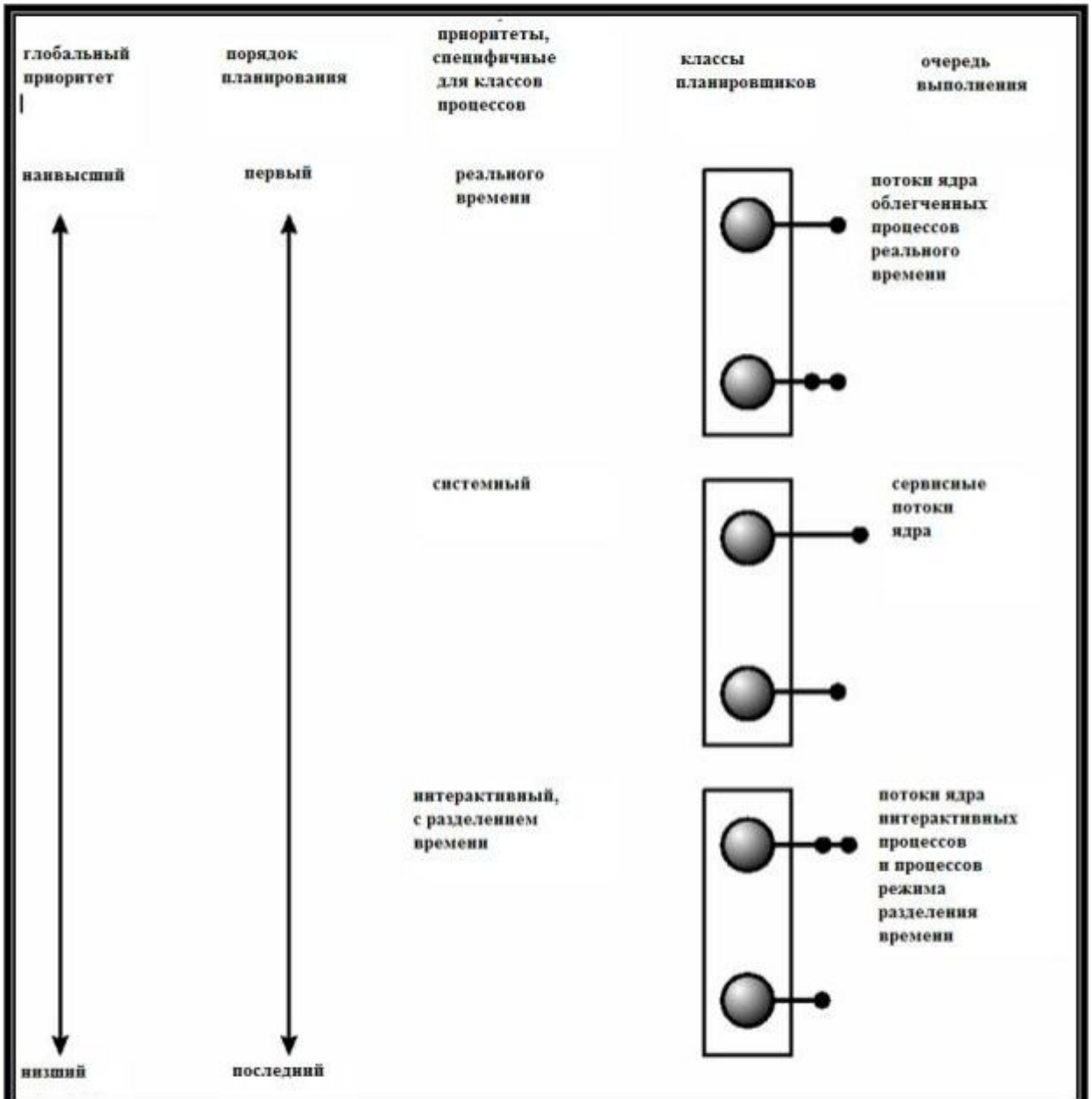
Планирование загрузки процессоров в реальном времени

- ***Системы реального времени класса 1 (hard real-time systems)*** **– требуют решения критической задачи за фиксированный интервал времени**
- ***Системы реального времени класса 2 (soft real-time computing)*****– требуют, чтобы критические процессы имели высший приоритет, по сравнению с обычными.**

Латентность диспетчера (dispatch latency)



Планирование в Solaris 2



Приоритеты в Windows

	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

Операционные системы

Методы синхронизации процессов.

История

- **Совместный доступ к общим данным может привести к нарушению их целостности (inconsistency).**
- **Поддержание целостности общих данных требует механизмов упорядочения работы взаимодействующих процессов (потоков).**
- **Решение проблемы общего буфера с помощью глобальной (общей) памяти допускает, чтобы не более чем *n* – 1 элементов данных могли быть записаны в буфер в каждый момент времени.**
  - **Предположим, что в системе производитель/потребитель мы модифицируем код, добавляя переменную *counter*, инициализируемую о и увеличиваемую каждый раз, когда в буфер добавляется новый элемент данных**

Ограниченный буфер

- **Общие данные**

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

- **Процесс-производитель**

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

- **Процесс-потребитель**

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

Ограниченный буфер

- **Операторы counter++;counter--;**
  должны быть выполнены атомарно (*atomically*).
- **Атомарная операция – такая, которая должна быть выполнена полностью без каких-либо прерываний. При этом, операция, выполняемая одним из процессов, является неделимой, с точки зрения другого процесса**

- **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

Ограниченный буфер

- **Если и производитель, и потребитель пытаются обратиться к буферу совместно (одновременно), то указанные ассемблерные операторы также должны быть выполнены совместно (interleaved).**
- **Реализация такого совместного выполнения зависит от того, каким образом происходит планирование для процессов – производителя и потребителя.**

- **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.**

Конкуренция за общие данные (race condition)

- **Race condition: Ситуация, когда взаимодействующие процессы могут обращаться к общим данным совместно (параллельно). Конечное значение общей переменной зависит от того, какой процесс завершится первым.**

- **Для предотвращения подобных ситуаций, процессы следует *синхронизировать*.**

Проблема критической секции

- ***n* процессов – каждый может обратиться к общим данным**
- **Каждый процесс имеет участок кода, называемый *критической секцией,* в котором происходит обращение к общим данным.**
- **Проблема – обеспечить, чтобы, если один процесс вошел в свою критическую секцию, никакой другой процесс не мог бы одновременно войти в свою критическую секцию.**

Решение проблемы критической секции

1. ***Взаимное исключение***. **Если процесс P_i исполняет свою критическую секцию, то никакой другой процесс не должен в тот же момент времени исполнять свою.**
2. ***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.**
3. ***Ограниченное ожидание***. **Должно существовать ограничение на число раз, которое процессам разрешено входить в свои критические секции, после того как некоторый процесс сделал запрос о входе в критическую секцию, и до того, как этот запрос удовлетворен.**

- **Предполагается, что каждый процесс исполняется с ненулевой скоростью**
- **Не делается никаких специальных предположений о соотношении скоростей каждого из *n* 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:**
  - **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 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);
```

- **Удовлетворяет принципу “взаимное исключение”, но не принципу “прогресс”**

Алгоритм 3

- **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.**

Алгоритм булочной (bakery algorithm)

Автор данного алгоритма – Л. Лампорт (L. Lamport). Рассмотрим алгоритм, решующий проблему синхронизации по критическим секциям. Происхождение названия следующее: алгоритм как бы воспроизводит стратегию автомата в (американской) булочной, где каждому клиенту присваивается его номер в очереди. В нашей российской реальности, данный алгоритм более уместно было бы назвать по этой же причине "алгоритм Сбербанка".

- **Обозначения: $<$ ≡ лексикографический порядок**
- **$(a,b) < (c,d)$ если $a < c$ or if $a = c$ and $b < d$**
  - **max $(a_0,\ldots, a_{n-1})$- число k, такое, что $k \geq a_i$ 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,j] < number[i,i])) ;
    }
    critical section
    number[i] = 0;
    remainder section
} while (1);
```

Аппаратная поддержка синхронизации

- **Атомарная операция проверки и модификации значения 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
}
```

Аппаратное решение для синхронизации

- Атомарная перестановка 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
}
```

Общие семафоры – counting semaphores (по Э. Дейкстре)

- Семафоры-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

- Определяем семафор как структуру:

```
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 о
- Код:

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

Два типа семафоров

- *Общий семафор (Counting* semaphore) – целое значение, теоретически неограниченное
- *Двоичный семафор (Binary* semaphore) – целое значение, которое может быть только 0 или 1; возможно, проще реализуется (семафорный бит – как в Burroughs и “Эльбрусе”)
- Общий семафор S может быть реализован с помощью двоичного семафора.

Вариант операции wait(S) для системных процессов (“Эльбрус”)

- Для системного процесса лишние прерывания нежелательны и может оказаться важным удерживать процессор за собой на какое-то время

- Операция *ЖУЖ(S);* (вместо *ЖДАТЬ(S);*) – процесс не прерывается и “жужжит” на процессоре, пока семафор S не будет открыт операцией *ОТКРЫТЬ*

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**

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);
```

Классические задачи синхронизации

- Задача “ограниченный буфер” (Bounded-Buffer Problem)
- Задача “читатели-писатели” (Readers and Writers Problem)
- Задача “обедающие философы” (Dining-Philosophers Problem)

Задача “ограниченный буфер”

- 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);
```

Семафор mutex используется "симметрично"; над ним выполняется пара операций: wait … signal – семафорные скобки. Его роль –чисто взаимное исключение критических секций. Семафор empty сигнализирует об исчерпании буфера. В начале он закрыт, так как элементов в буфере нет. Поэтому при закрытом семафоре empty потребитель вынужден ждать. Открывает семафор empty производитель, после того, как он записывает в буфер очередной элемент. Семафор full сигнализирует о переполнении буфера. В начале он равен n – максимальному числу элементов в буфере. Производитель перед записью элемента в буфер выполняет операцию wait (full), гарантируя, что, если буфер переполнен, записи нового элемента в буфер не будет. Открывает семафор full потребитель, после того, как он освободил очередной элемент буфера.

Задача “читатели-писатели”

- Общие данные:
  semaphore mutex, wrt;

  Начальные значения:
  mutex = 1, wrt = 1, readcount = 0

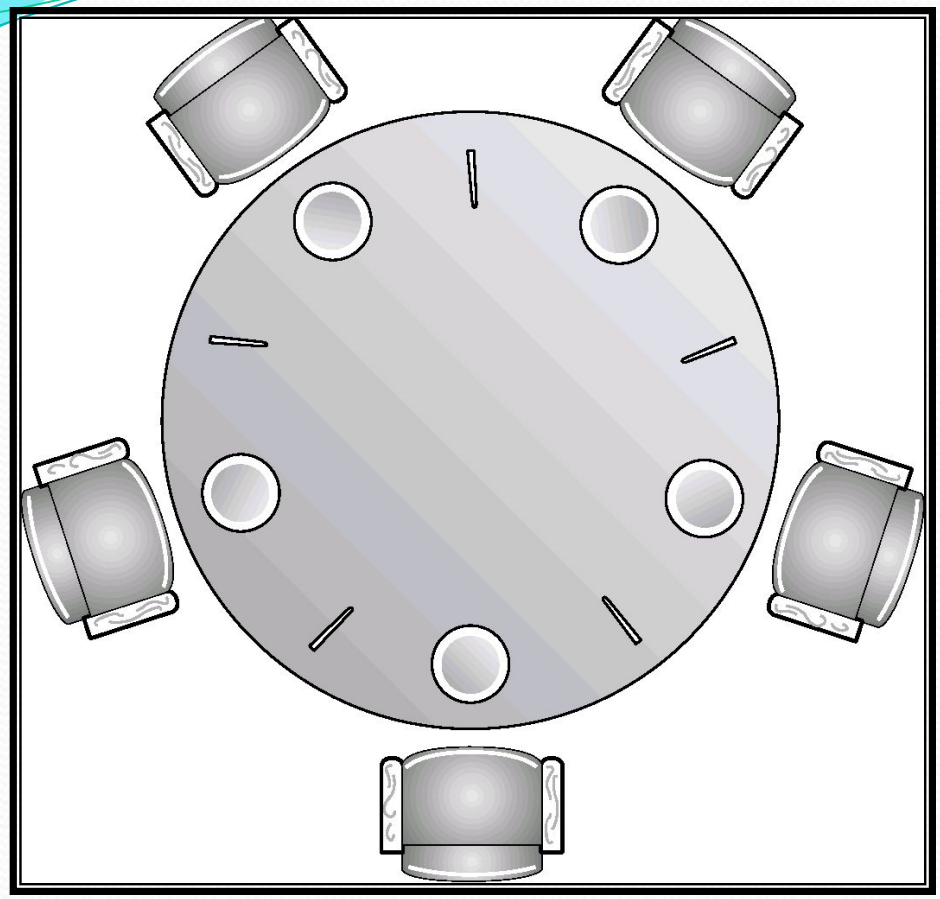
Процесс-писатель

```
wait(wrt);
        ...
    выполняется зап
        ...
signal(wrt);
```

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”



Суть задачи обедающие философы в следующем. 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.

- **Общие данные**

  semaphore chopstick[5];

  Первоначально все значения равны 1

Задача “обедающие философы”

- Философ *i*:

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

Критические области (critical regions)

- Высокоуровневая конструкция для синхронизации
- Общая переменная ***v*** типа ***T***, определяемая следующим образом:

  v: shared T
- К переменной ***v*** доступ возможен только с помощью специальной конструкции:

  region v when B do S

  где ***B*** – булевское выражение.

- Пока исполняется оператор ***S***, больше ни один процесс не имеет доступа к переменной ***v***.

Пример: ограниченный буфер

- Общие данные:

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

Процесс-производитель

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

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

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*.

Реализация

- Число процессов, ждущих на **first-delay** и **second-delay**, хранится, соответственно, в **first-count** и **second-count**.

- Алгоритм предполагает упорядочение типа FIFO процессов в очереди к семафору.

- Для произвольной дисциплины обслуживания очереди требуется более сложный алгоритм.

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:**

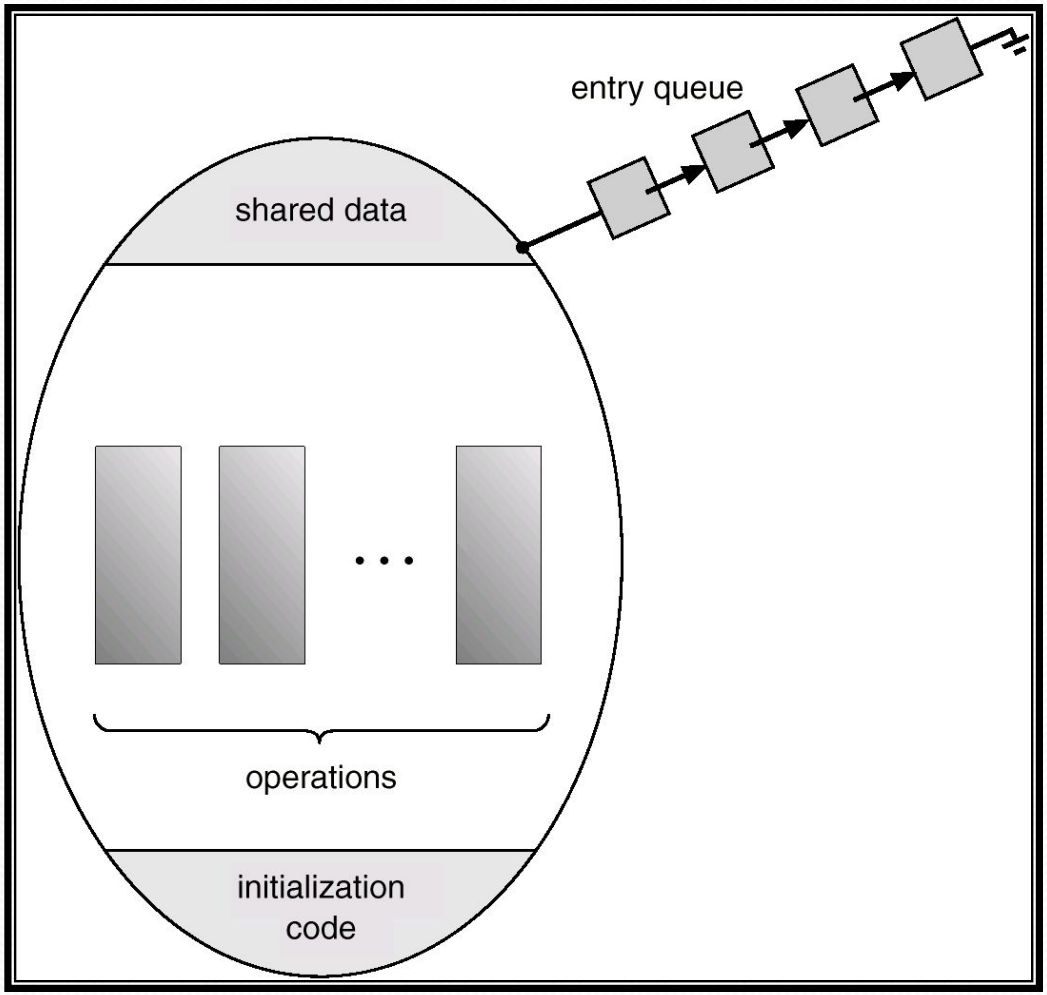
  condition x, y;
- **Условные переменные могут использоваться только в операциях wait и signal.**
  - **Операция:**

    x.wait();

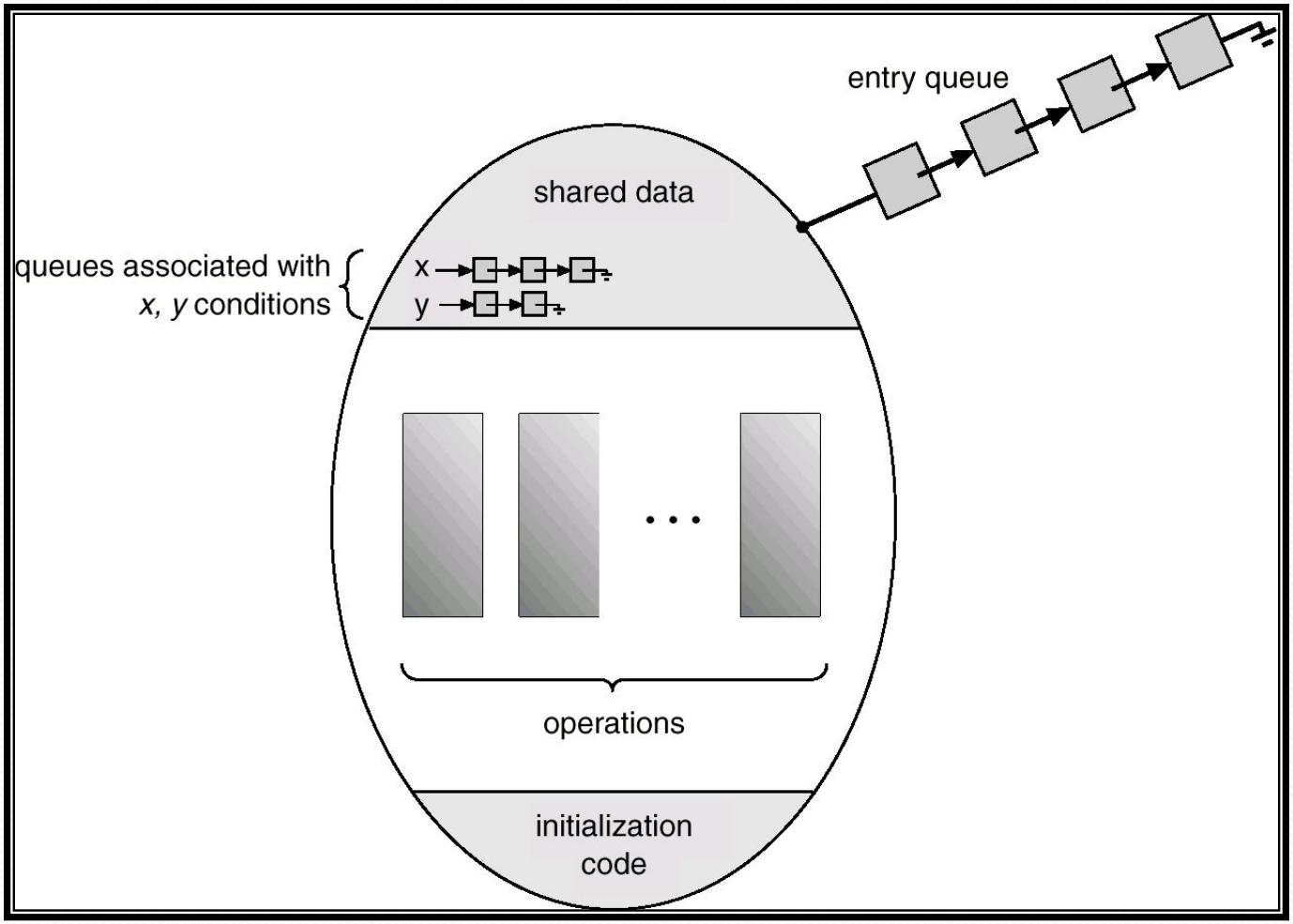
    означает, что выполнивший ее процесс задерживается до того момента, пока другой процесс не выполнит операцию:

    x.signal();
  - **Операция x.signal возобновляет ровно один приостановленный процесс. Если приостановленных процессов нет, эта операция не выполняет никаких действий.**

Схематическое представление монитора



Монитор с условными переменными



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

- 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--;
}
```

Реализация мониторов

- Конструкция *conditional-wait*: **x.wait(c);**
  - **c** – целое выражение, вычисляемое при исполнении операции **wait**.
  - значение **c** (*приоритет*) сохраняется вместе с приостановленным процессом.
  - когда исполняется **x.signal**, первым возобновляется процесс с меньшим значением приоритета .
- Для обеспечения корректности работы системы проверяются два условия:
  - Пользовательские процессы должны всегда выполнять вызовы операций монитора в правильной последовательности.
  - Необходимо убедиться, что никакой процесс не игнорирует вход в монитор и не пытается обратиться к ресурсу непосредственно, минуя протокол, предоставляемый монитором .

Синхронизация в Solaris 2

- Реализованы разнообразные виды блокировок для поддержки многозадачности, многопоточности (включая потоки реального времени) и мультипроцессирования.
- Используются *adaptive mutexes* в целях повышения эффективности для защиты данных (при обработке их короткими сегментами кода).
- Используются *condition variables* и *readers-writers locks* (для более длинных сегментов кода).
- Используются *turnstiles* (“вертушки”) для реализации списка потоков, ожидающих либо адаптивного mutex, либо reader-writer lock.

Синхронизация в Windows

- **Для защиты доступа к данным на однопроцессорных системах используются маски прерываний.**
- **Используются spinlocks для многопроцессорных систем.**
- **Также обеспечиваются *dispatcher objects*, которые могут работать как mutex'ы или семафоры.**
- **Dispatcher objects генерируют события (*events*). Семантика события аналогична семантике условной переменной.**