



# Операционные системы

Курс лекций для гр. 4057/2

## **Лекция №3**

# Вопросы к лекции 2

1. В многопроцессорной системе производительность возрастает всегда при назначении потоков разным процессам. А при каких условиях возрастает производительность многопоточного процесса в однопроцессорной системе?
2. При запуске MS Word порождаются три потока; далее их число может возрастать. Предложите, какие части Word разумно выполнять как отдельные потоки.
3. Сравните достоинства и недостатки этого подхода с двумя альтернативными: а) все клиенты обслуживаются одним процессом, б) для каждого клиента рождается отдельный процесс (для примера возьмите веб-сервер).
4. Какими должны быть потоки сервера для каждого клиента – потоками ядра или пользователя?
5. В Windows 3.1 (1991 г.) была невытесняющая диспетчеризация. В чем заключаются недостатки такого "неполноценного" мультипрограммирования ?
6. Нарисуйте диаграмму переключения состояний прикладного процесса в MS-DOS.
7. Для каких типов вычислительных систем какие критерии более приоритетны? (Рассмотрите три типа систем: пакетной обработки, интерактивные системы и реального времени.)
8. Докажите (путем простого рассуждения) указанную оптимальность алгоритма SJF.
9. Предложите *простой* способ *приближенного* предсказания этого времени на основе предыстории активности процесса.
10. В чем недостатки статических приоритетов ?

## Вопросы к лекции 2 (2)

11. Как с помощью динамических приоритетов можно предотвращать голодание низкоприоритетных процессов?
12. Заполните таблицы в предположении нулевого времени переключения контекста и отсутствия ввода-вывода. Процессы находятся в очереди готовых в порядке возрастания номеров и стартуют одновременно. Длительность кванта - 1 сек.

Процесс №	Длина, сек.	Время обслуживания		Время ожидания	
		FCFS	RR	FCFS	RR
1	10				
2	10				
3	10				
4	10				
Среднее:					

## Вопросы к лекции 2 (2)

11. Как с помощью динамических приоритетов можно предотвращать голодание низкоприоритетных процессов?
12. Предложите способ предотвращения инверсии приоритетов
13. Заполните таблицы в предположении нулевого времени переключения контекста и отсутствия ввода-вывода. Процессы находятся в очереди готовых в порядке возрастания номеров и стартуют одновременно. Длительность кванта - 1 сек.

Процесс №	Длина, сек.	Время обслуживания		Время ожидания	
		FCFS	RR	FCFS	RR
1	10				
2	10				
3	10				
4	10				
Среднее:					

## Вопросы к лекции 2 (3)

Процесс №	Длина, сек.	Время обслуживания			Время ожидания		
		FCFS	RR	SJF	FCFS	RR	SJF
1	40						
2	30						
3	20						
4	10						
Среднее:							

# Содержание

## Раздел 1. Управление процессами

<...>

1.6 Процессы, потоки и их диспетчеризация в Windows, Unix и Linux

1.7 Диспетчеризация процессов в ОС реального времени

# Windows: процессы и потоки

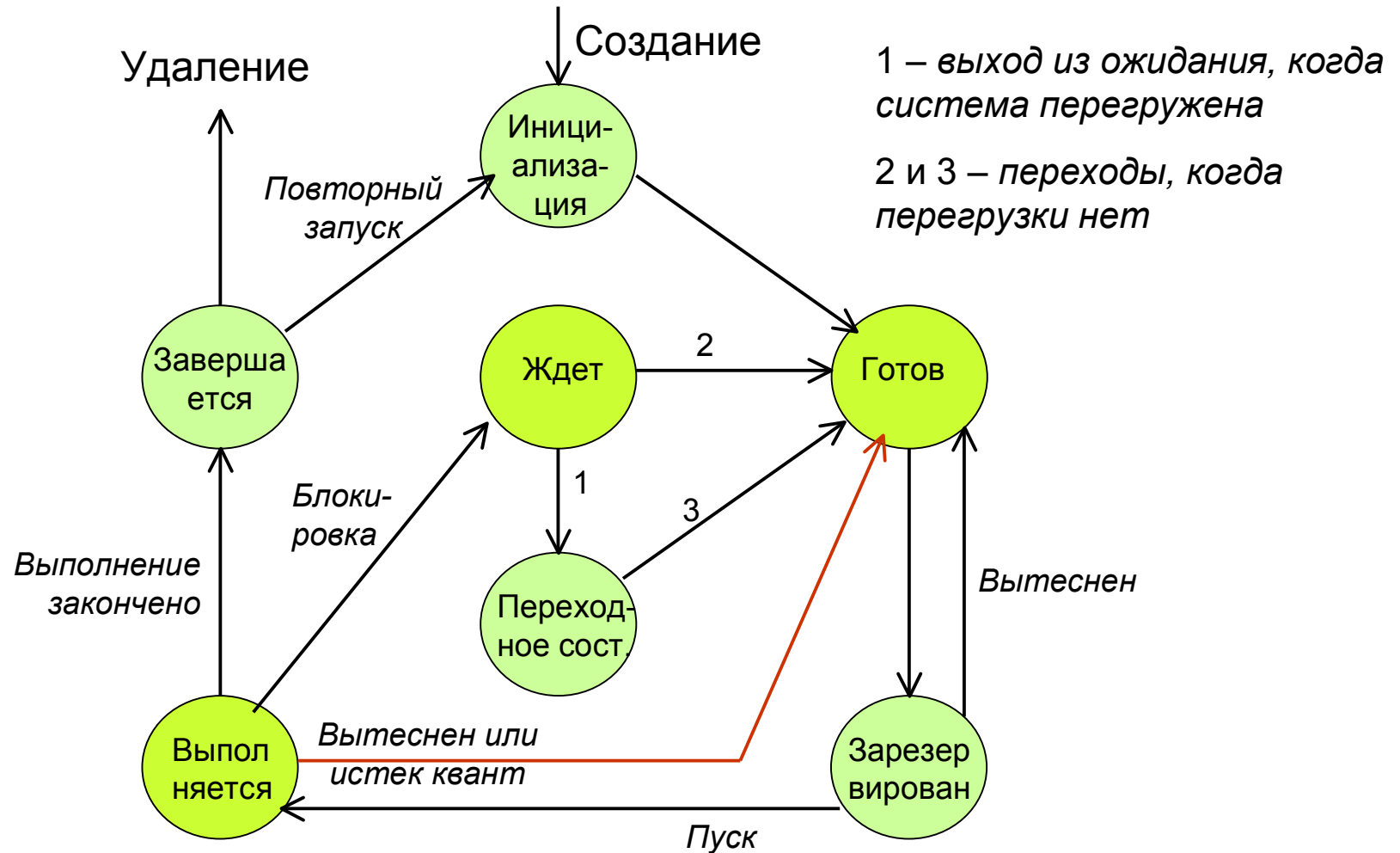
- Процесс – это контейнер набора ресурсов, используемых потоками, которые выполняют экземпляр программы. Он включает в себя:
  - виртуальное адресное пространство (ВАП)
  - исполняемую программу – код и данные, отображаемые на ВАП
  - список открытых системных ресурсов: файлов, коммуникационных портов, семафоров и пр.
  - контекст защиты – маркер доступа (access token), определяющий пользователя, группы безопасности и привилегии процесса
  - уникальный идентификатор процесса
  - как минимум, один поток; без него программа процесса не может выполняться
- Поток включает в себя:
  - аппаратный контекст – состояние процессора
  - два стека – один для режима ядра и другой – для пользовательского режима
  - локальную память потока (TLS) – область памяти, используемую библиотеками периода выполнения (run-time libraries) и DLL
  - уникальный идентификатор потока

# Windows: диспетчеризация потоков (1)

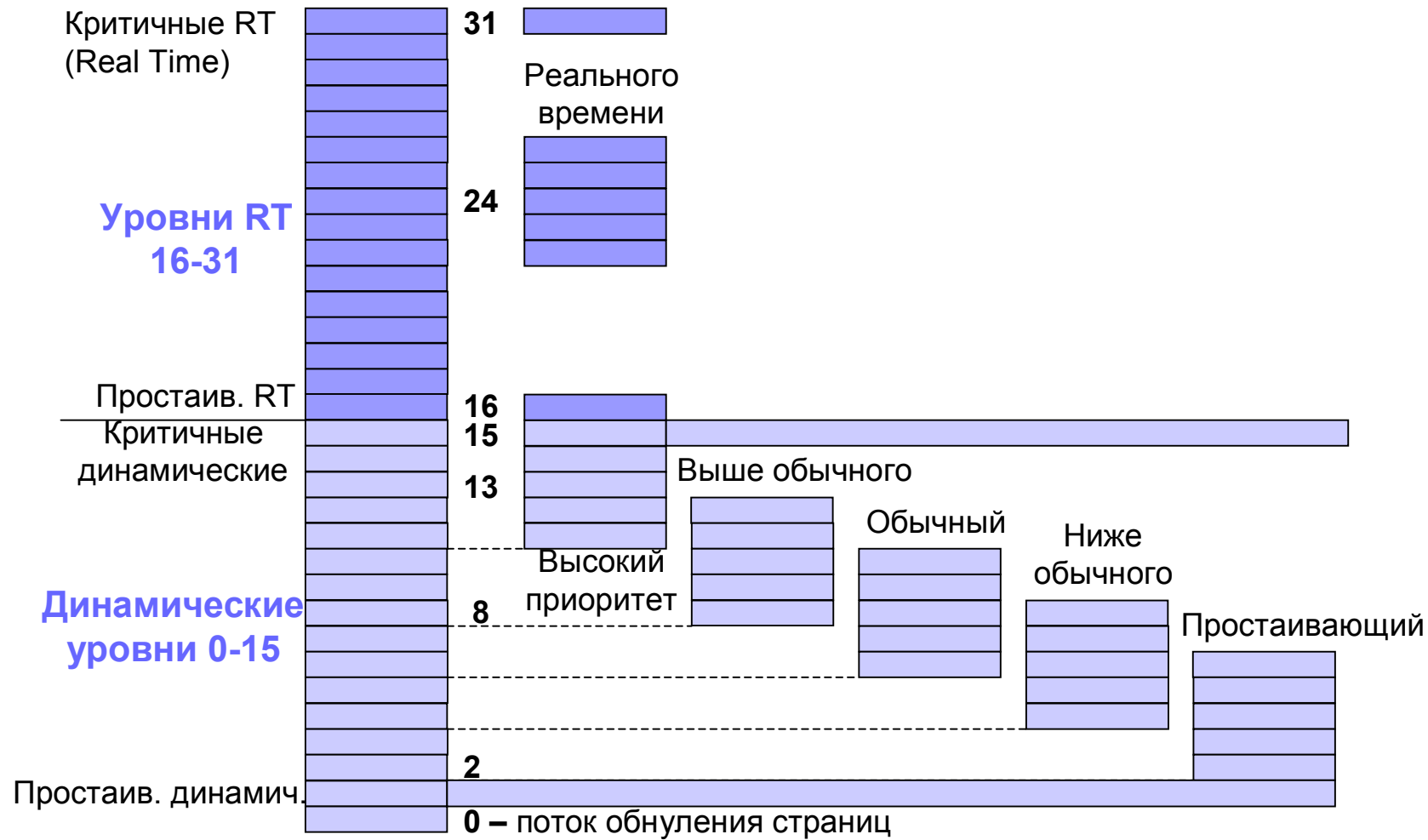
- Планирование осуществляется на уровне потоков; ОС не обращает внимания на то, какому процессу принадлежит поток. Сами процессы не выполняются, а лишь предоставляют ресурсы и контекст для выполнения своих потоков (Вопрос 1)
- Планирование – вытесняющее: как только в состояние готовности переходит поток с более высоким приоритетом, чем текущий, то последний вытесняется, даже если его квант еще не истек, и помещается в начало очереди готовых данного приоритета
- Возможны потоки ядра и потоки пользователей
- Потоки могут порождать дочерние потоки – **волокна (fibers)**. Когда волокно блокируется, то оно помещается в очередь заблокированных, а для работы выбирается другое волокно в контексте того же потока
  - таким образом снимается следующий недостаток потоков пользователей: блокирование одного потока приводит к ожиданию остальных в том же процессе (см. лекцию 2, слайд 11)
- При перегрузке системы (напр., нехватке памяти) поток переводится в переходное состояние – не участвует в конкуренции за время процессора



# Windows : диаграмма состояний ПОТОКОВ



# Windows: приоритеты потоков



# Windows : управление приоритетами

- API-функция *SetPriorityClass* устанавливает класс приоритета процесса при его создании; базовый приоритет процесса по умолчанию равен среднему значению внутри класса
- Большинство приложений имеет «Обычный» класс приоритета
- Потоки первоначально наследуют базовый приоритет процесса, далее их *текущий* приоритет можно изменять:
  - устанавливать функцией *SetThreadPriority* относительный приоритет внутри класса (5 уровней)
  - повышать в динамическом диапазоне от 1 до 15 функцией *SetThreadPriorityBoost*
- В диапазоне RT (16-31) текущий приоритет потока всегда = базовому; **статические приоритеты** делают планирование потоков более предсказуемым
- У каждого потока свое значение кванта; по умолчанию начальная величина кванта = 20 мс
- ОС может изменять величины квантов активных процессов динамически; их возможные значения: 20, 40, 60, 80, 120 мс

# Windows: диспетчеризация потоков (2)

- Набор структур данных для планирования – *база данных диспетчера ядра* (dispatcher database)
- Основная структура – *очередь готовых потоков*
- Алгоритм диспетчеризации – MLFB:
  - при выходе из состояния ожидания приоритет потока повышается на величину от 1 до 8, в зависимости от вида увв-вы
  - далее он уменьшается на 1 с каждым окончанием кванта
  - никогда не опускается ниже базового приоритета
- Голодающие потоки – те, кто в состоянии готовности более 3 сек; для их обнаружения системный поток (диспетчер настройки баланса) раз в 1 сек сканирует очередь готовых потоков
  - Приоритет обнаруженного голодающего потока повышается до 15 и он получает удвоенный квант; затем приоритет снижается до исходного уровня
  - Это, в частности, способ борьбы с инверсией приоритетов

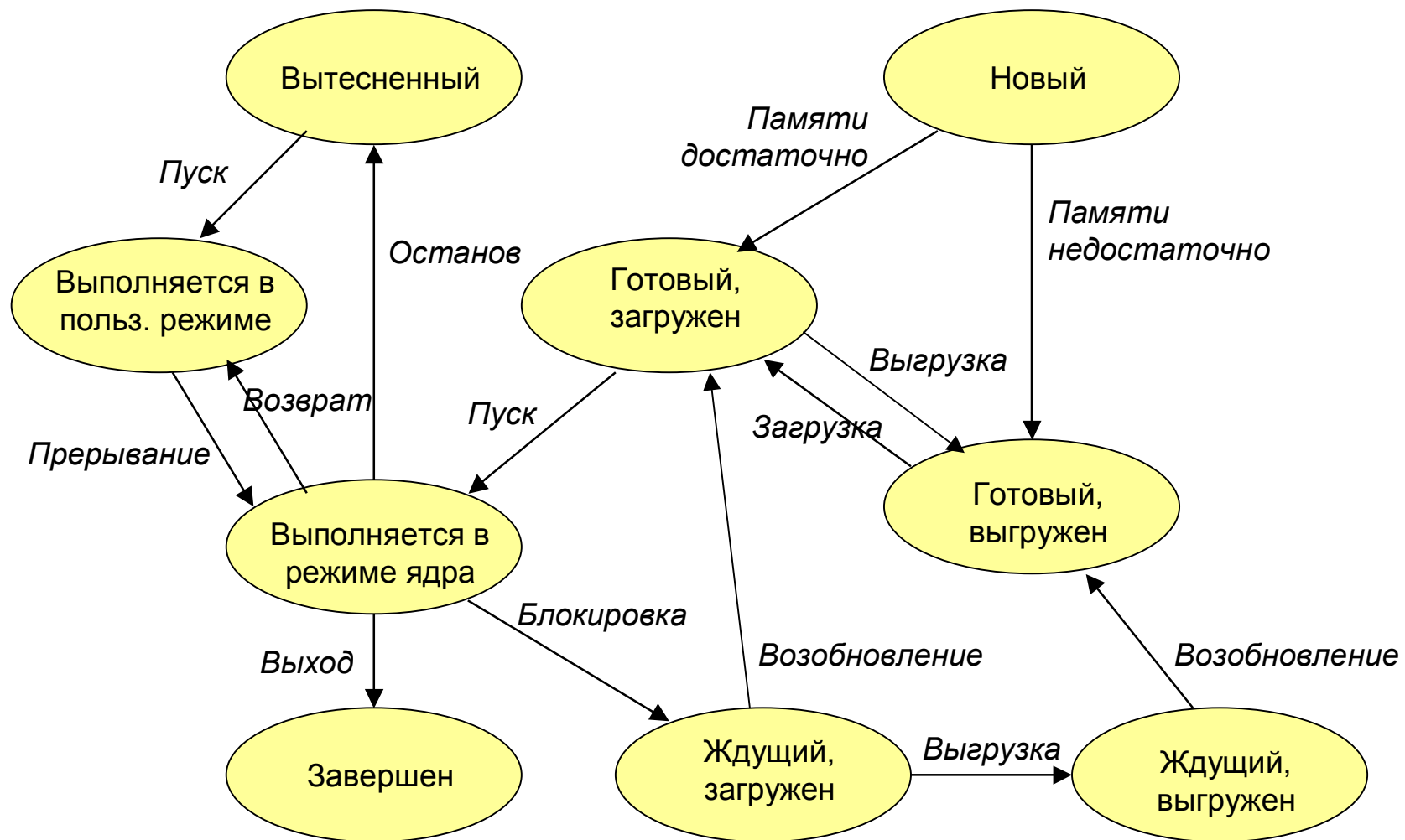
# Unix: процессы и потоки

- Ранние версии Unix не имели потоков
- В UNIX BSD возможны только потоки пользователей
- В UNIX SystemV и Solaris (как и в Linux) потоки поддерживаются ядром
  - в ряде случаев возникают неопределенные ситуации – результат столкновения концепций процесса и потока

## Основные отличия от Windows

- Процессы – активные сущности
- Ждущие и готовые процессы могут быть выгружены на диск. Этот процесс *свопинга* управляется планировщиком верхнего уровня
- Большая часть программ Unix выполняются в контексте пользовательских процессов, а Windows основные функции ядра организованы как отдельные системные процессы
  - поэтому в результате системного вызова или прерывания поток переходит в режим ядра с полным доступом к ресурсам машины

# Диаграмма состояний процесса в UNIX System 5



# Unix: диспетчеризация

- У процессов в режиме пользователя (целые) значения приоритетов  $> 0$ , в режиме ядра  $\leq 0$ ; чем меньше значение, тем выше приоритет.
- Вариант MLFB: раз в секунду приоритет  $P$  каждого процесса пересчитывается по формуле:  $P = base + nice + CPU\_usage$ , где:
  - $base$  – начальное значение, привязывающее процесс к некоторой группе
  - $20 < nice \leq -20$  – статическое смещение приоритета внутри группы (пользователь может задавать только  $nice > 0$ )
  - $CPU\_usage$  – *взвешенное среднее* время использования процессора (CPU порция)

# Unix: усреднение CPU порции

- *Взвешенное*, или *скользящее* усреднение – обычный способ предсказания будущего значения на основе прошлых значений
- Здесь вычисляется  $S_{n+1}$  - предсказание  $T_{i+1}$  - длительности следующей CPU порции
- Простейшая формула – обычное среднее:  $S_{n+1} = \frac{1}{n} \sum_{i=1}^n T_i$
- Чтобы не вычислять сумму повторно, можно представить это так:

$$S_{n+1} = \frac{1}{n} T_n + \frac{n-1}{n} S_n$$

- Чтобы увеличить вес последних измерений, применяют формулу «старения»:  $S_{n+1} = a T_n + (1 - a) S_n$ ,  
где  $a$  – постоянный весовой множитель ( $0 < a < 1$ )
- Простейшая реализация:  $a = 1/2$ ; на каждом шаге прибавляется новое значение и результат сдвигается на 1 бит вправо
- В разных версиях UNIX «прошиты» различные значения  $a$



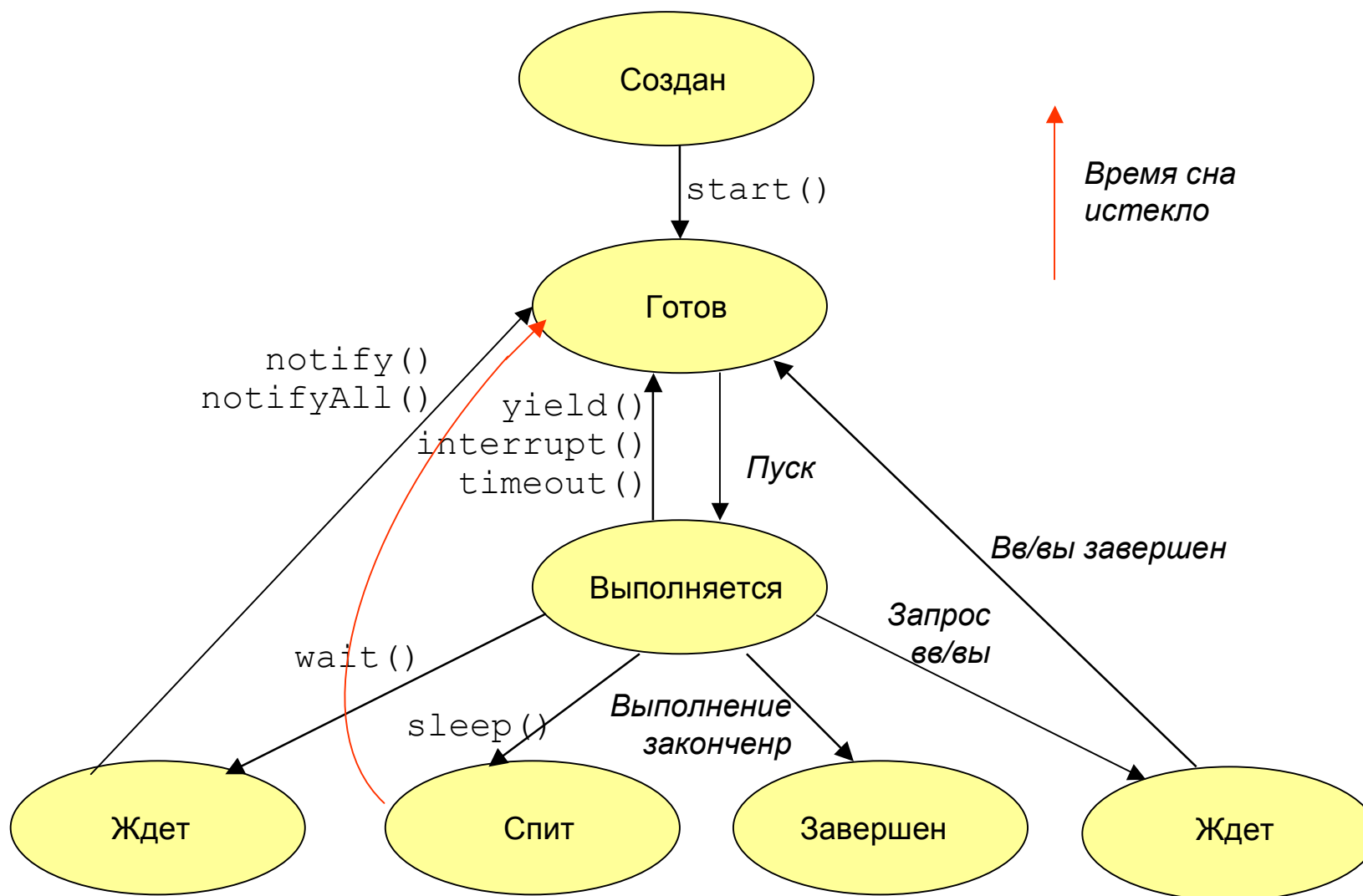
# Linux: потоки

- Потоки создаются вызовом `clone(function, stack_ptr, sharing_flags, arg)`; в зависимости от флага `sharing_flags`:
  - либо в текущем процессе
  - либо в новом процессе (т.е., как `fork`)
- Новый поток начинает выполнение функции `function` с параметрами `arg`
- Биты массива `sharing_flags` описывают, какие ресурсы (файлы, таблицы обработчика сигналов и т.д.) - общие для родительского и сыновнего потока
- Есть три класса задач в порядке убывания приоритетов:
  - Реального времени – FCFS (вытесняющая дисциплина)
  - Реального времени – RR с вытеснением
  - Разделения времени – RR без вытеснения
- Значения приоритетов и длительности квантов всех потоков пересчитываются при каждом вызове диспетчера – вариант MLFB
- Многие версии ядра Linux не различают процессы и потоки
- В диаграмме состояний потока нет свопинга, но есть приостановка (`stop/continue`)

# Java: потоки

- Это потоки пользователей, порождаемые программой
- JVM поддерживает многопоточную вытесняющую диспетчеризацию с RR для потоков одинакового приоритета
  - в некоторых реализациях JVM вытеснения нет
- Приоритет потока – целое число от 1 до 10; по умолчанию 5
- Приоритет может назначаться методом `setPriority(int newPriority)`
- Метод `yield()` – приостановка текущего потока для того, чтобы процессор был выделен другому потоку того же приоритета
  - Вызовы `yield()` вставляют в код для гарантии выполнимости приложения на платформе без вытеснения

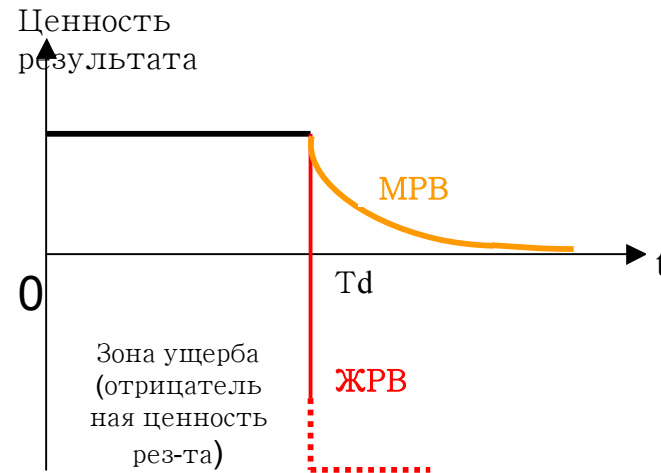
# Java: диаграмма состояний потоков



# Диспетчеризация процессов в ОС реального времени

Два вида задач РВ:

- Жесткого РВ (Hard RT): требуется *гарантированное* время отклика  $\leq$  заданного срока  $T_d$  (deadline)
- Мягкого РВ (Soft RT): требуется *минимально возможное* время отклика, т.е. процессы РВ должны просто выполняться с приоритетом перед остальными, а превышение заданных сроков допустимо, хотя и нежелательно



Под временем отклика  $t$  понимается промежуток времени от запроса на выполнение процесса до выдачи результата

# Характеристики задач РВ

- **ЖРВ:** управление техническими системами (транспорт, вооружение, сети связи, технологическое оборудование и т.п.), часто реализованное на *встроенных (embedded)* микропроцессорах и компьютерах - например, одноплатных (бездисковых) РС

ОС для ЖРВ обычно имеют малый размер, многозадачность, вытесняющую приоритетную диспетчеризацию

Что облегчает задачу планирования процессов ЖРВ:

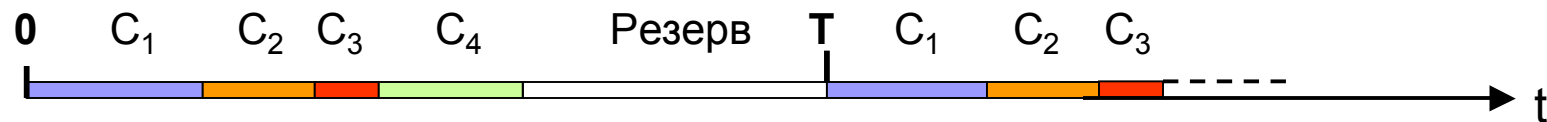
- фиксированный набор программ с заранее известными временными характеристиками → упрощает назначение приоритетов и квантов времени (возможно статическое планирование)
- часто нет виртуальной памяти → сокращается и становится предсказуемым время переключения контекста

- **МРВ:** интерактивная графика, мультимедиа, некоторые системы коммуникации. Даже ОС общего назначения иногда могут поддерживать МРВ-приложения

# Диспетчеризация процессов РВ

Два вида процессов РВ:

- *периодические* – реагирующие на сигналы, возникающие через фиксированные промежутки времени (например, сигналы датчиков при периодическом их опросе)
- *непериодические* – на сигналы, возникающие в непредсказуемые моменты времени (например, сигнал аварии)
- Диспетчеризация периодических процессов проще
- Для нескольких процессов с *одинаковым* периодом  $T$  и предельными сроками  $\geq T$  подходит даже невытесняющий FCFS алгоритм
  - в этом случае необходимое и достаточное условие выполнимости требований ЖРВ:  $\sum C_i \leq T$  ( $i=1, \dots, n$ ), где  $C_i$  - время выполнения запроса  $i$ -го процесса на процессоре:



# Диспетчеризация периодических процессов РВ

- Если процессы имеют *неодинаковые* периоды, то ни FCFS, ни RR в общем случае не гарантируют выполнение предельных сроков
  - «Наложение» различных периодов создает время от времени пиковые ситуации большой нагрузки
  - Поэтому чем больше резерв времени процессора, тем больше вероятность выполнения предельных сроков
- Гарантии выполнимости дает частотно-монотонный алгоритм (*Rate Monotonic Scheduling, RMS*):
  - вытесняющая RR дисциплина
  - процесс с более коротким периодом имеет более высокий статический приоритет
  - При условии, что требуемая CPU порция у каждого из N процессов постоянна на всех интервалах повторения, доказано (1973), что RMS обеспечивает выполнимость сроков ЖРВ, если процессы потребляют в среднем U - долю времени CPU:

$$U \leq N(2^{1/N} - 1)$$

- Например, для N=3  $U \leq 0,78$  ; при  $N \rightarrow \infty$   $U \rightarrow \ln 2 \sim 0,693...$ 
  - Эта формула – не для запоминания, а пример интересного теоретического результата
  - NB: если процессы с более короткими периодами требуют меньшие CPU порции, то алгоритм MLFB автоматически работает как RMS

# Диспетчеризация неперiodических процессов РВ

- Для неперiodических процессов гарантию выполнения предельных сроков дает алгоритм ближайшего срока (**Earliest Deadline Algorithm, EDA**):
  - вытесняющая дисциплина
  - динамические приоритеты: чем раньше предельный срок отклика среди готовых процессов, тем выше приоритет процесса
- EDA работает даже при 100% средней загрузке процессора, т.е. при  $U = 1$ 
  - при этом требуемая CPU порция у каждого из  $N$  процессов не обязательно постоянна при каждом запросе, как требует RMS
- Таким образом, EDA эффективнее, чем RMS, но и сложнее

**NB:** Разработчики стремятся проектировать системы РВ как системы периодических процессов со статическими приоритетами, как более простые

- Для неперiodических сигналов производится регулярный опрос состояния → процессы их обработки становятся периодическими



# Диспетчеризация в конкретных ОС РВ

В **QNX** возможны 3 дисциплины:

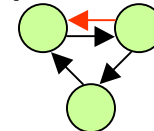
1. Многоуровневые очереди (MLQ) FIFO с приоритетами (возможно, динамическими)
2. То же, но RR на каждом уровне приоритетов
3. Адаптивная (Adaptive scheduling) – следующий вариант MLFB:
  - Процессу назначается начальный (базовый) приоритет
  - Если процесс израсходовал квант времени, но не блокировался (и если есть готовый процесс того же приоритета), то его приоритет понижается на 1
  - Если готовый процесс не получает времени процессора в течение 1 сек, то его приоритет повышается на 1, но не выше базового
  - В момент блокировки процесс повышает свой приоритет до базового

В **pSOS** диспетчеризация MLFB с RR

- Временно можно отключать квантование и запрещать вытеснение
- Базовый приоритет устанавливается приложением и может быть изменен ОС
  - Например, в ситуации инверсии приоритета (см. слайд 24 лекции 2) производится *наследование приоритета*: приоритет процесса L увеличивается до приоритета H и возвращается к исходному после освобождения неразделяемого ресурса

# Выводы

- Все современные ОС поддерживают приоритетные алгоритмы диспетчеризации MLQ
- Базовая диаграмма состояний процессов дополняется в различных ОС промежуточными состояниями и приостановкой / свопингом
- Активные сущности (т.е., процессы в полном смысле слова):
  - в Windows – потоки и волокна
  - в Unix/Linux – процессы и потоки
  - в Java – потоки программы пользователя
- Требования к реактивности (т.е., ко времени отклика) в системах жесткого и мягкого реального времени существенно различны
- Для диспетчеризации периодических процессов РВ популярен частотно-монотонный алгоритм (статические приоритеты), для непериодических – алгоритм ближайшего срока (динамические приоритеты)
- Термин «Потоки РВ» в Windows можно понимать только условно, поскольку время переключения контекста велико и отсутствуют динамические приоритеты



# Вопросы к лекции 3

- Для задач МРВ