Операционные системы

Курс лекций для гр. 4057/2

Лекция №3

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

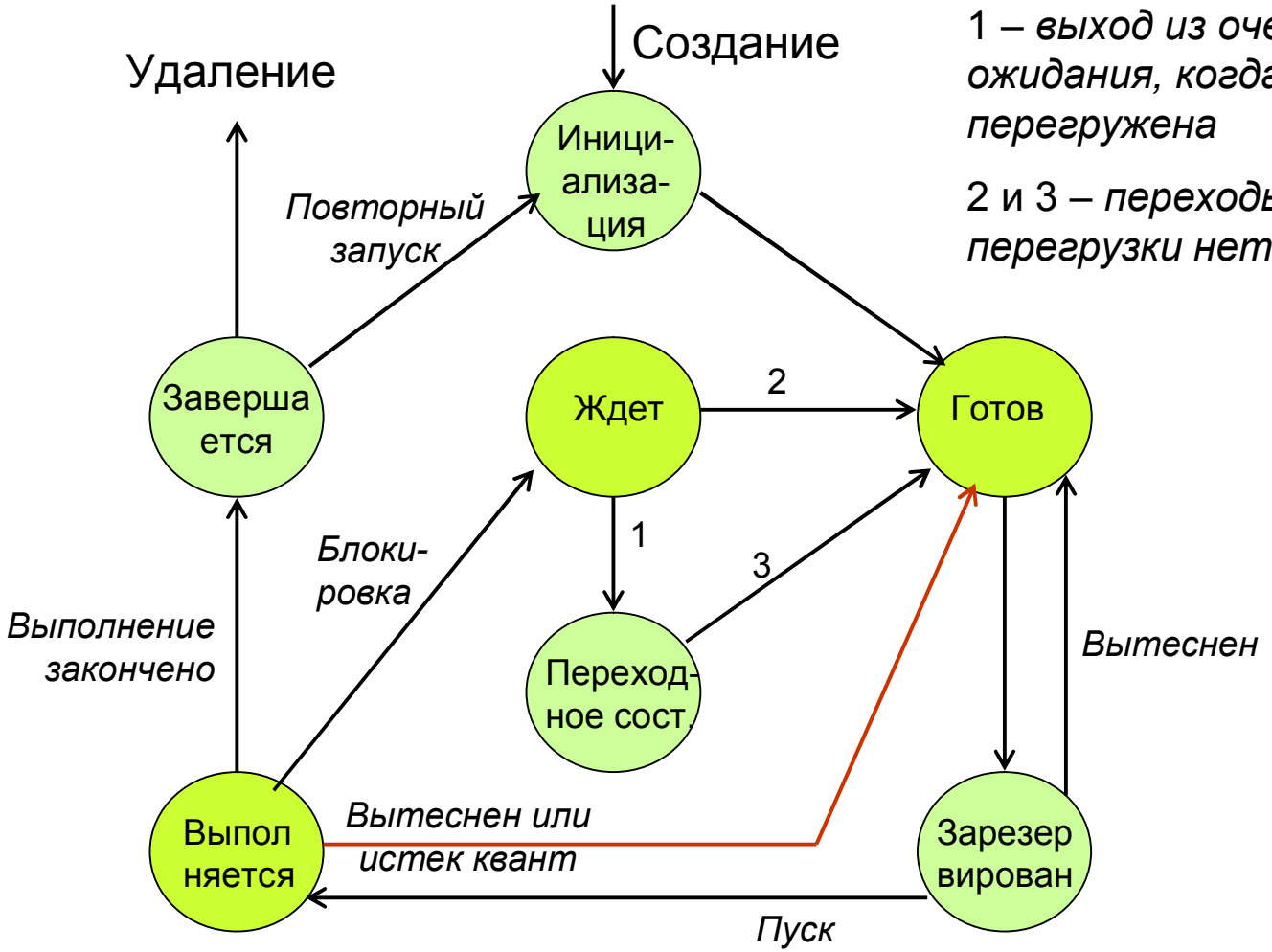
Windows: процессы и потоки

- 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 блокированных, 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

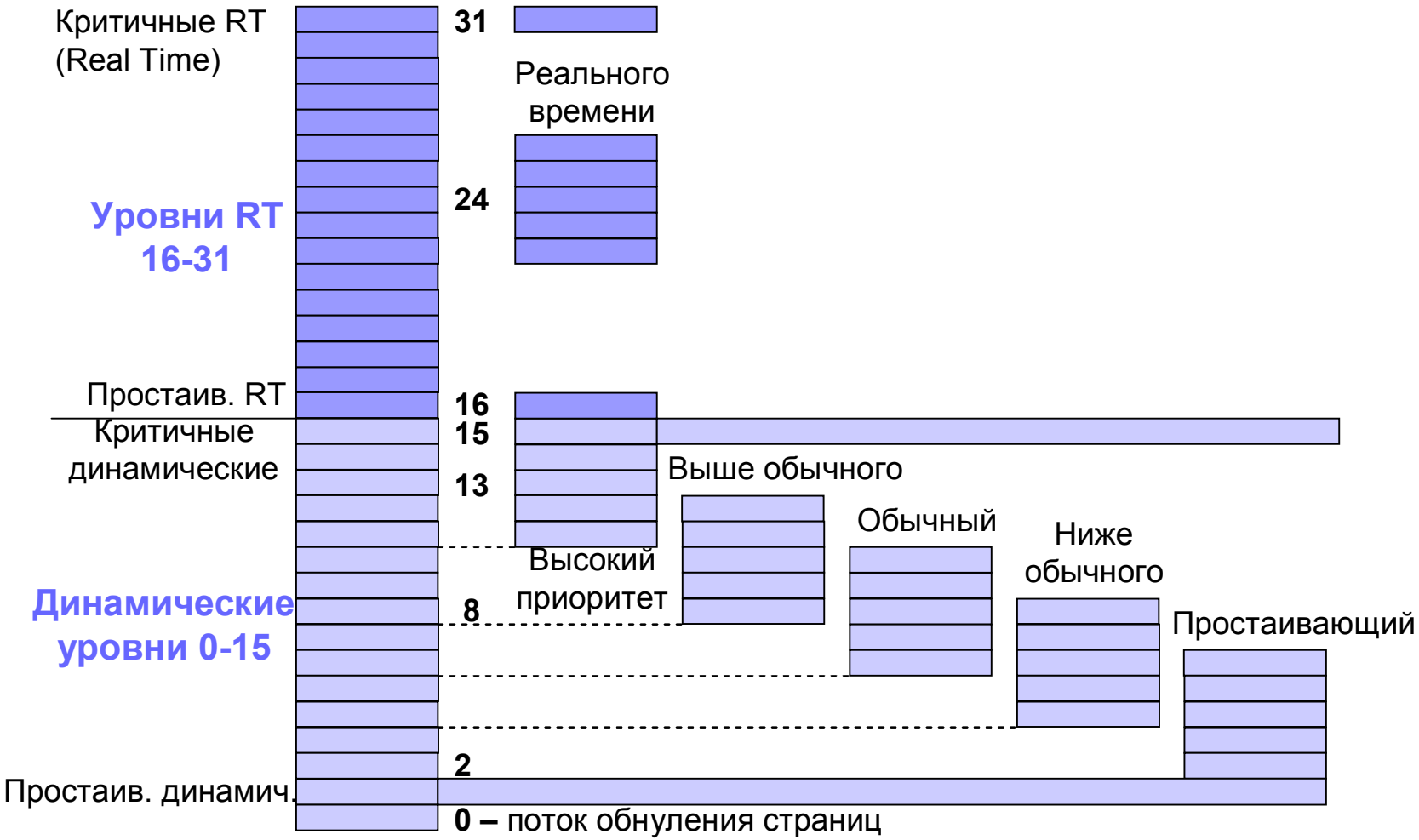
Windows : диаграмма состояний потоков



1 – *выход из очереди ожидания, когда система перегружена*

2 и 3 – *переходы, когда перегрузки нет*

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

  (Вопрос 2)

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 долее 3 сек; для их обнаружения системный поток (диспетчер настройки баланса) раз в 1 сек сканирует очередь готовых потоков
  - Приоритет обнаруженного голодающего потока повышается до15 и он получает удвоенный квант; затем приоритет снижается до исходного уровня
  - Это, в частности, способ борьбы с инверсией приоритетов

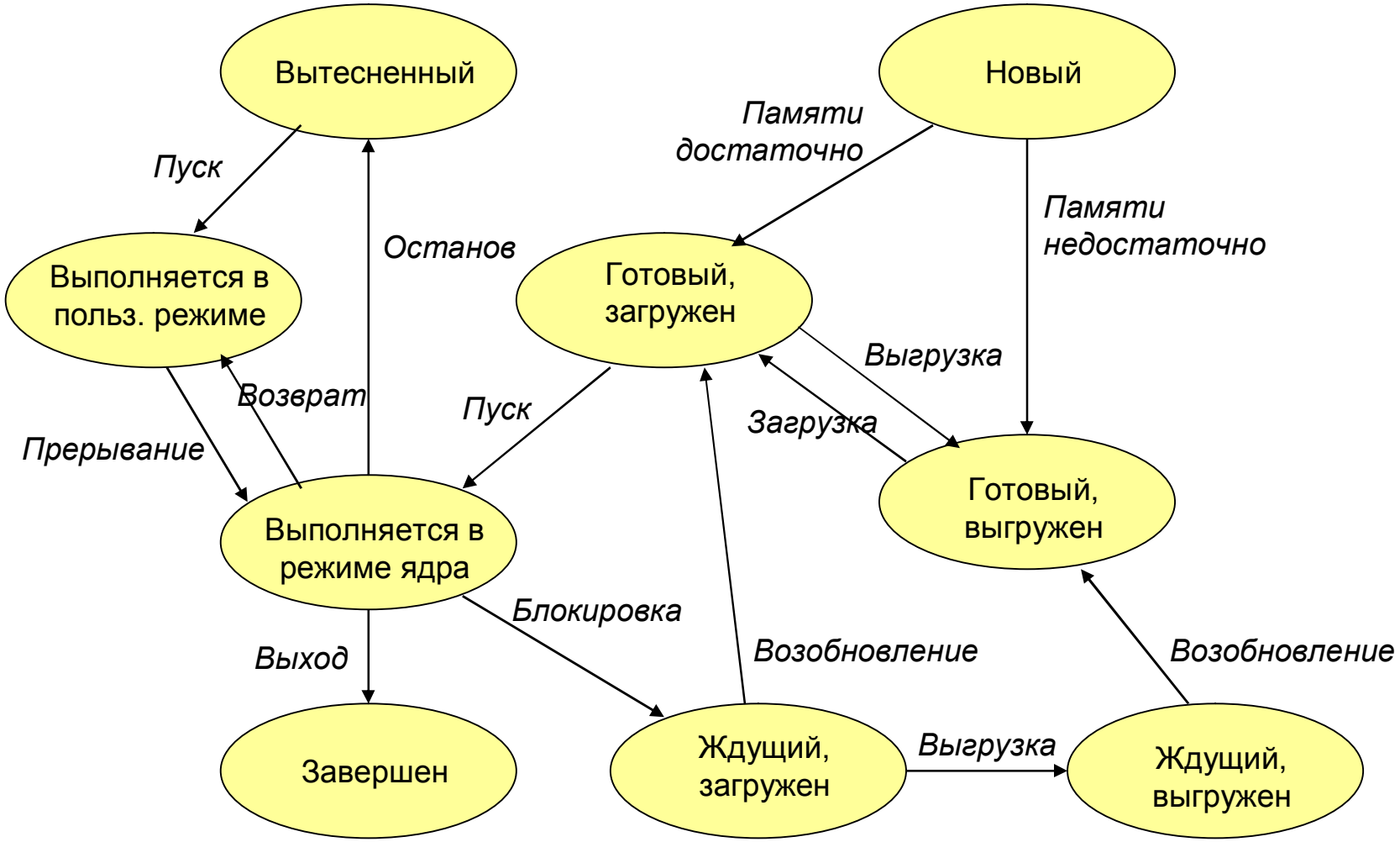
Unix/Linux: процессы и потоки

- Ранние версии Unix не имели потоков
- В UNIX BSD возможны только потоки пользователей
- В UNIX SystemV, Solaris и в Linux) потоки поддерживаются ядром
  - в ряде случаев возникают неопределенные ситуации – результат столкновения концепций процесса и потока (Вопрос 3)

Основные отличия от Windows

- Процессы – активные сущности
- Ждущие и готовые процессы могут быть выгружены на диск. Этот процесс *свопинга* управляется планировщиком верхнего уровня
- Б**о**льшая 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/Linux



Unix/Linux: диспетчеризация

- У процессов в режиме пользователя (целые) значения приоритетов > 0, в режиме ядра ≤ 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* ≤ -20 – статическое смещение приоритета внутри группы (пользователь может задавать только *nice* > 0)
  - *CPU_usage* – *взвешенное среднее* время использования процессора (CPU порция)

Unix/Linux: 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

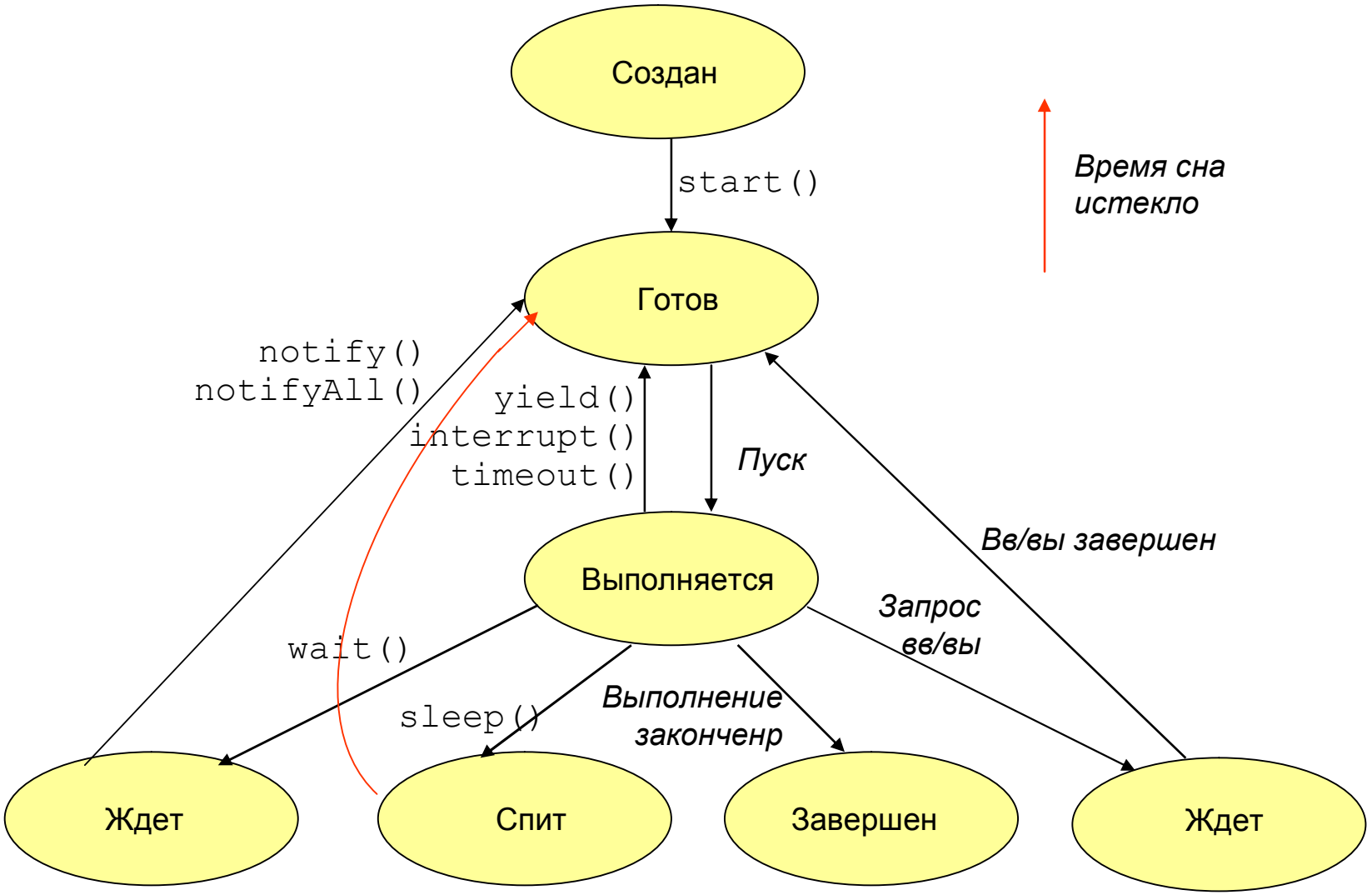
- Потоки создаются вызовом clone(function, stack_ptr, sharing_flags, arg); в зависимости от флага sharing_flags:
  - либо в текущем процессе
  - либо в новом процессе (т.е., как fork)
- Новый поток начинает выполнение функции function с параметрами arg
- Биты масссива sharing_flags описывают, какие ресурсы (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 квантования 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 квантования

(Вопрос 4)

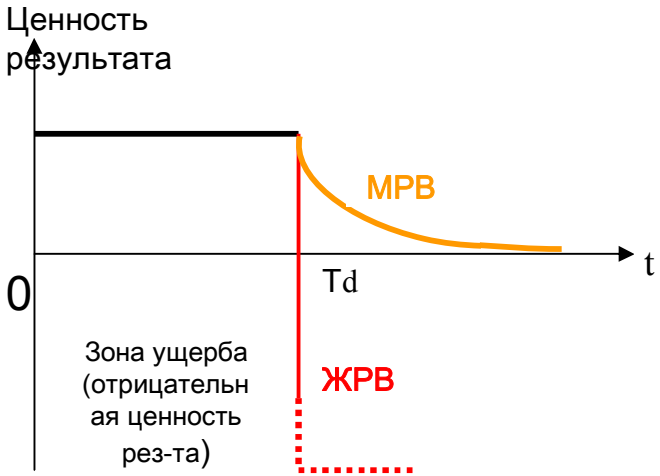
Java: диаграмма состояний потоков



Диспетчеризация процессов в ОС реального времени

Два вида задач РВ:

- Жесткого РВ (Hard RT): требуется *гарантированное* время отклика ≤ заданного срокаTd (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) PC

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

Что облегчает задачу планирования процессов ЖРВ:

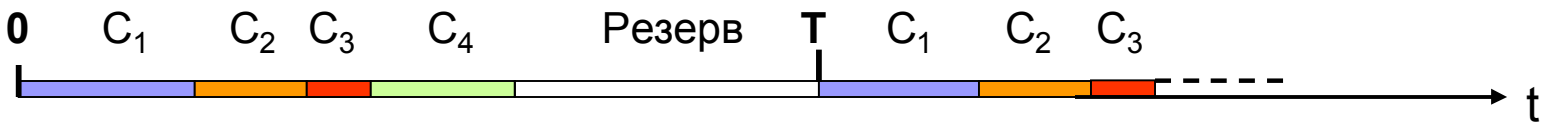
- фиксированный набор программ с заранее известными временными характеристиками → упрощает назначение приоритетов и размеров квантов времени (возможно статическое планирование)
- часто нет виртуальной памяти → сокращается и становится предсказуемым время переключения контекста

- **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 >= T подходит даже невытесняющий FCFS алгоритм
  - в этом случае необходимое и достаточное условие выполнимости требований ЖРВ: $\sum C_i \le T$ (i=1,...n), где C_i - время выполнения запроса i-го процесса на процессоре:



(Вопрос 5)

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 в срок 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 $N \rightarrow \infty$ $U \rightarrow \ln 2 \sim 0{,}693\ldots$
    - 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 непериодических процессов РВ

- Для непериодических процессов гарантию выполнения предельных сроков дает алгоритм ближайшего срока (Earliest Deadline First, EDF):
  - вытесняющая дисциплина
  - динамические приоритеты: чем раньше предельный срок отклика среди готовых процессов, тем выше приоритет процесса
- EDF 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, EDF эффективнее, чем RMS, но сложнее и менее предсказуемый

  (Вопрос 6)

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

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

Диспетчеризация в конкретных ОС РВ

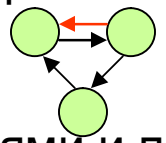
В **QNX** возможны 3 дисциплины:

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 гарантию выполнимости в срок дает частотно-монотонный алгоритм RMA (статические приоритеты), для непериодических – алгоритм ближайшего срока EDF (динамические приоритеты)
- Термин «Потоки РВ» в Windows можно понимать только условно, поскольку время переключения контекста велико и отсутствуют динамические приоритеты

Вопросы к лекции

1. Пусть у процесса A есть 10, а у процесса B - 2 готовых к выполнению потоков, и все 12 имеют одинаковый приоритет. Какую долю процессорного времени получит каждый из потоков при круговой диспетчеризации ?
2. Удлинение кванта потока – еще один способ дать ему преимущество в конкуренции за время процессора. Сравните этот способ с повышением приоритета потока: как он влияет на реактивность и производительность?
3. Рассмотрите случай процесса с несколькими потоками, один из которых блокирован - ожидает ввода с клавиатуры. Предположим, этот процесс выполняет fork. Должна ли копия блокированного потока тоже быть блокирована? Если да, то какому из двух потоков адресовать вводимую строку? Если нет, то что должен делать этот поток в новом процессе?
4. При выполнении метода yield()процессор передается готовому потоку того же приоритета, что и у текущего, если таковой есть; иначе продолжается текущий поток. Почему так происходит – почему невозможна передача управления потоку другого приоритета?
5. Сформулируйте аналогичное условие выполнимости для случая N процессов с *различными* периодами Ti. (i = 1, …N), совпадающими с предельными сроками, и известными временами выполнения на процессоре Ci. Примените его к задаче: на обработку трех процессов с периодами 100, 200 и 500 мс уходит по 50, 30 и 100 мс соответственно.
6. Алгоритм EDF не учитывает разное время выполнения запроса после передачи управления процессу. Какую характеристику работы алгоритма можно улучшить, если в момент диспетчеризации T известно не только предельное время отклика Di (deadline), но и оставшееся время выполнения каждого i-го процесса Ci ? Предложите соответствующую модификацию EDF.