Java для систем реального времени

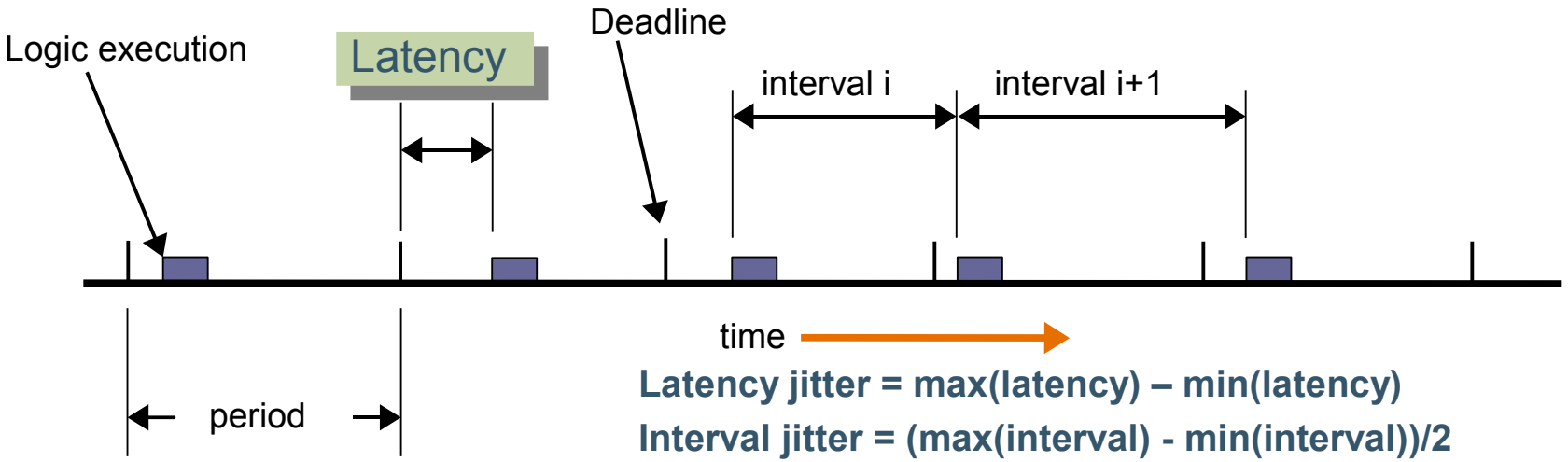
Sun Microsystems
Ekaterina.Pavlova@Sun.COM

Что означает “Real-time”?

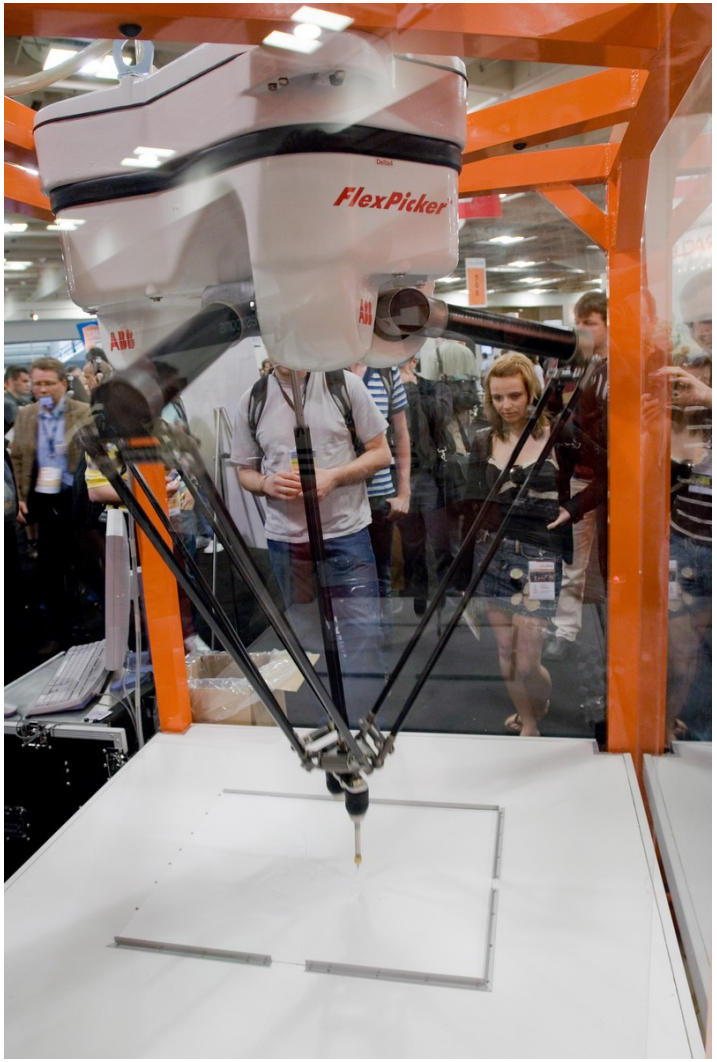
- Программа должна удовлетворять временным ограничениям
  - „Что” так же важно, как и „когда”
  - Результат полученный поздно — неправильный результат
- “Real-time” не значит “really fast”
  - От системы требуется выполнение задач в предсказуемые моменты времени
- Различают hard и soft real-time

Примеры временных ограничений

- **Deadline** — момент времени, до которого должно быть выполнено задание
- **Latency** — время отклика (от происхождения события до начала его обработки)
- **Jitter** — максимальный разброс времени отклика



Примеры Real-Time систем



Где используются Real-Time системы?

- Вооружённые силы, аэронавтика
- Телекоммуникационная инфраструктура
  - VoIP, PBX, сотовая связь
- Финансовые системы
  - QoS для клиентов
- Промышленность
  - Автоматизация и управление процессом производства
  - Энергетическая промышленность

Зачем нужна Java для разработки Real-Time приложений?

Зачем Java для Real-Time?

- Те же ответы, что и на вопрос „Зачем нужна Java?”, но в приложении к системам реального времени
  - Традиционные приложения на C/C++/ассемблере трудно разрабатывать, отлаживать и поддерживать
- Требования к системам реального времени растут
  - Увеличиваются как размер, так и сложность

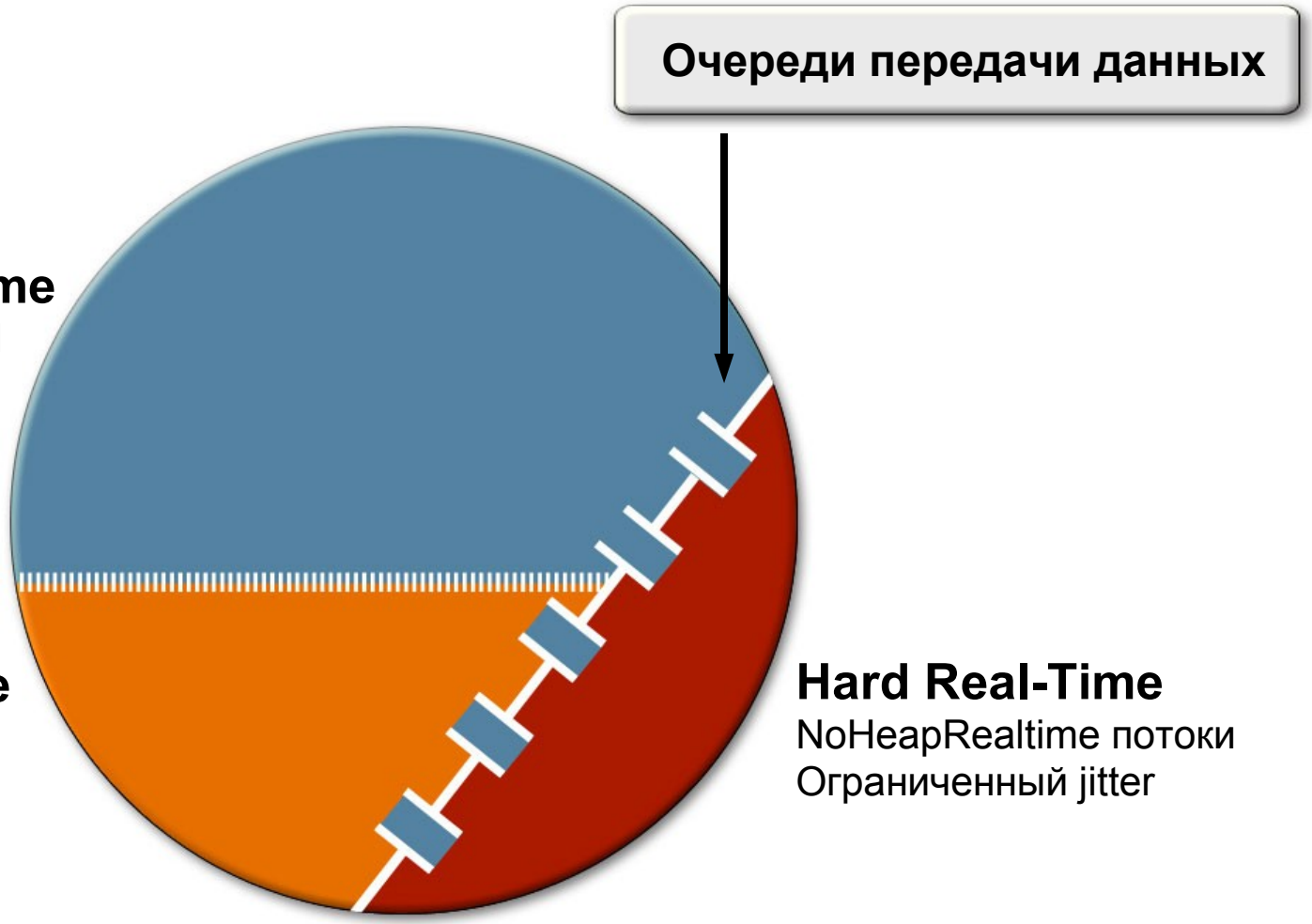
Предсказуема ли Java?

- Непредвиденные задержки:
  - Сборка мусора
  - Зависимость времени создания объектов от содержимого heap
  - Just-in-time компиляция
  - Динамическая загрузка классов
  - Возможна инверсия приоритетов
- Отсутствие прямого доступа к памяти
- Неоптимальное поведение планировщика задач

Real-Time Specification for Java

- JSR-001 и JSR-282
  - Определяет, как real-time поведение должно реализовываться при помощи Java-технологий
- Описывает API и семантические изменения Java платформы
  - Предоставляет высокоуровневые, переносимые абстракции
  - Возможность писать 100% Java кода

Модель системы по RTSJ



*RTGC не специфицирован RTSJ

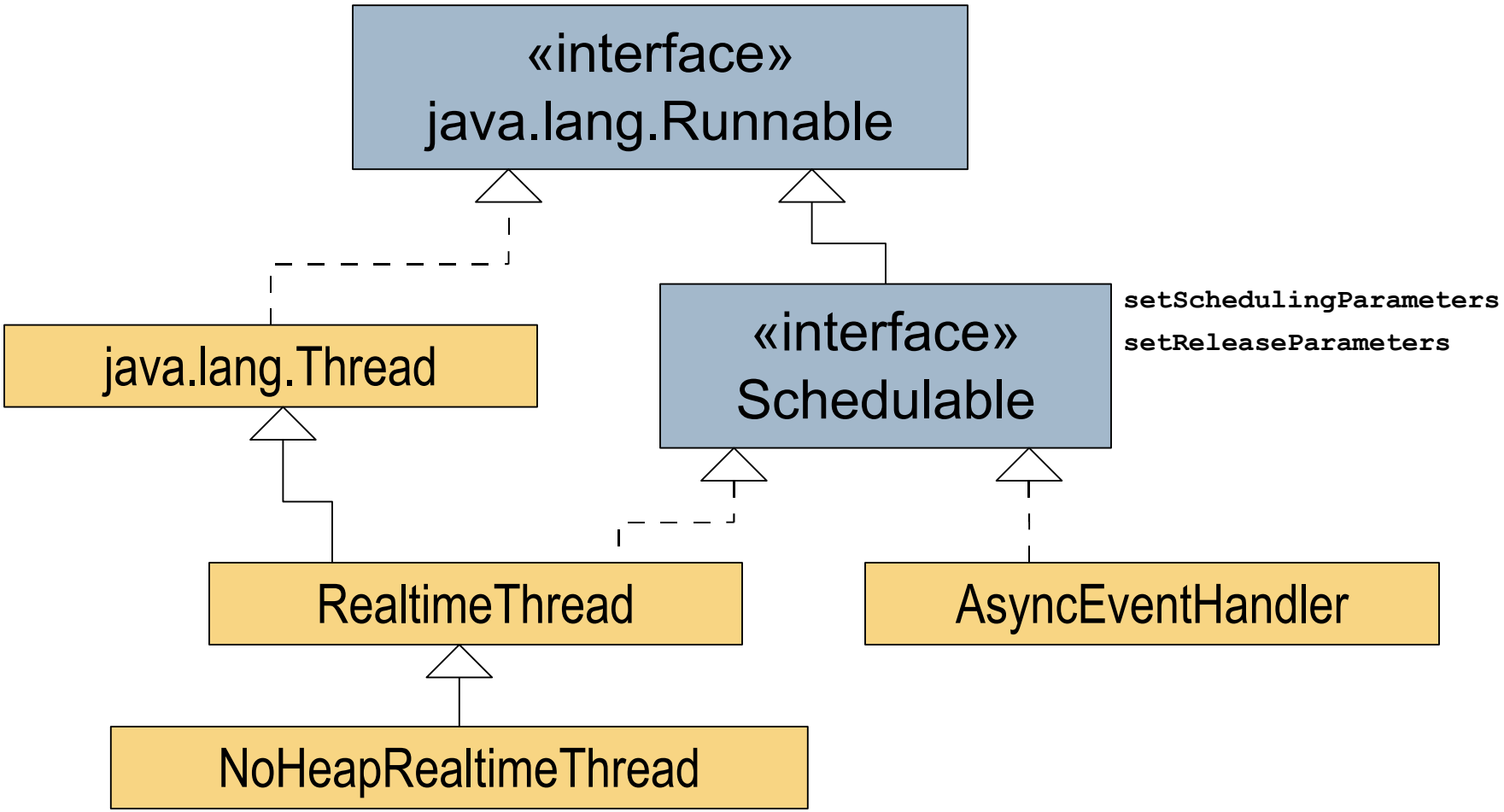
Java Real-Time System

- Реализация RTSJ компанией Sun Microsystems
  - Обратно совместима с Java SE 5
- Java RTS 2.1
  - Основана на Java SE 5u13
  - Работает на Solaris (SPARC и x86/x64) и на Linux x86 (SuSe, RHEL)
  - Использует встроенные real-time возможности на Solaris, RT-ядро на Linux
- Реализует Real-Time Garbage Collector

Ключевые возможности RTSJ

- Планирование и диспетчеризация
  - управление schedulable-объектами
- Синхронизация
  - ликвидация инверсии приоритетов
- Управление памятью
  - альтернативы heap-памяти
- Асинхронные события и обработчики
- Часы реального времени, таймеры

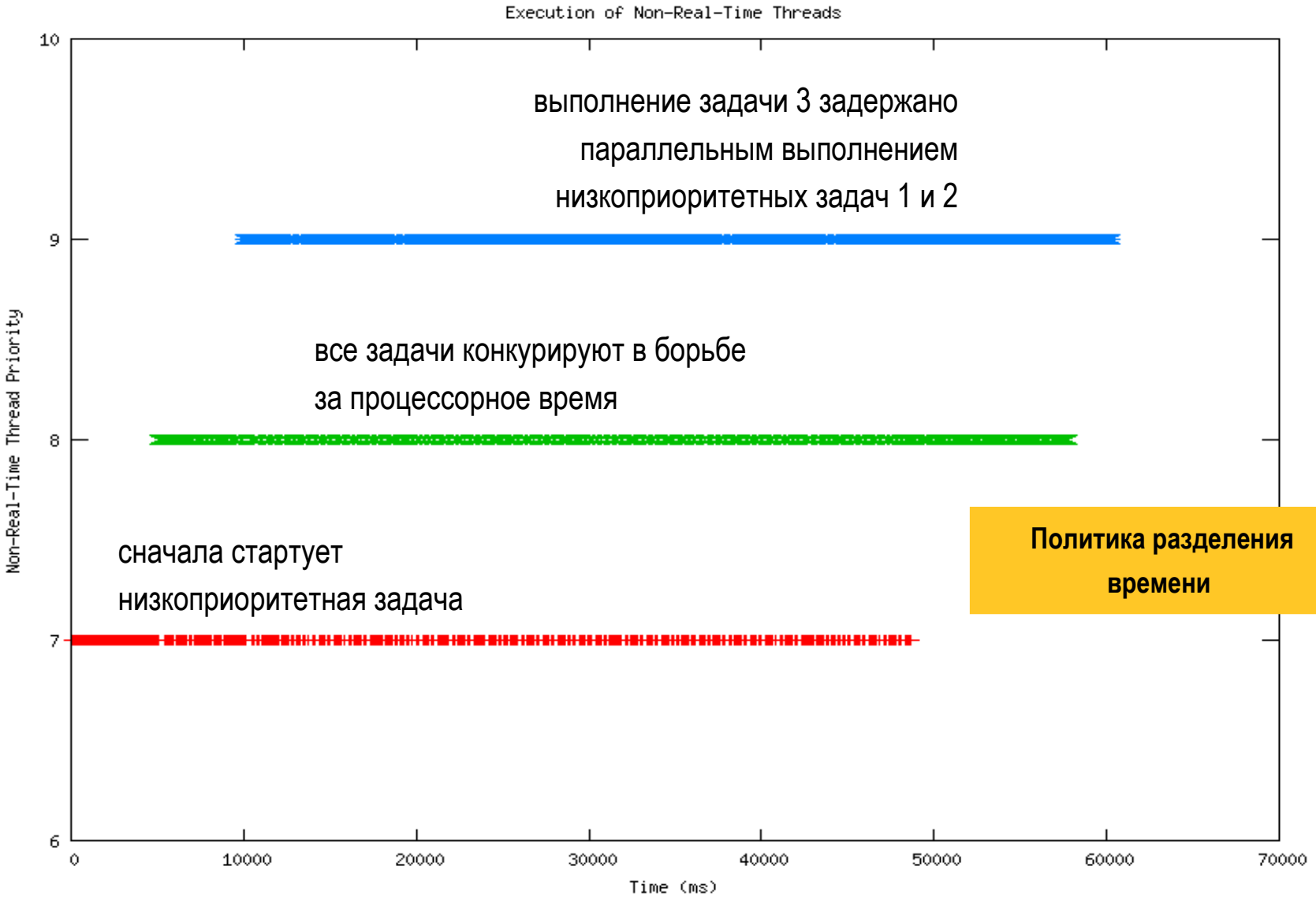
Потоки и schedulable-объекты



Планировка выполнения потоков

- Рассмотрим следующий набор задач:
  - Задача 1, низкий приоритет
    - стартует в момент t_0
  - Задача 2, средний приоритет
    - стартует в момент t_0+5 секунд
  - Задача 3, высокий приоритет
    - стартует в момент t_0+10 секунд
  - Все задачи требуют 20 секунд процессорного времени
  - Один процессор
- Вопросы
  - Какая задача завершится первой?
  - Когда завершится каждая из задач?

Диспетчеризация в Java SE



Политика разделения времени

Программирование в Java RTS, шаг 1

- Заменим

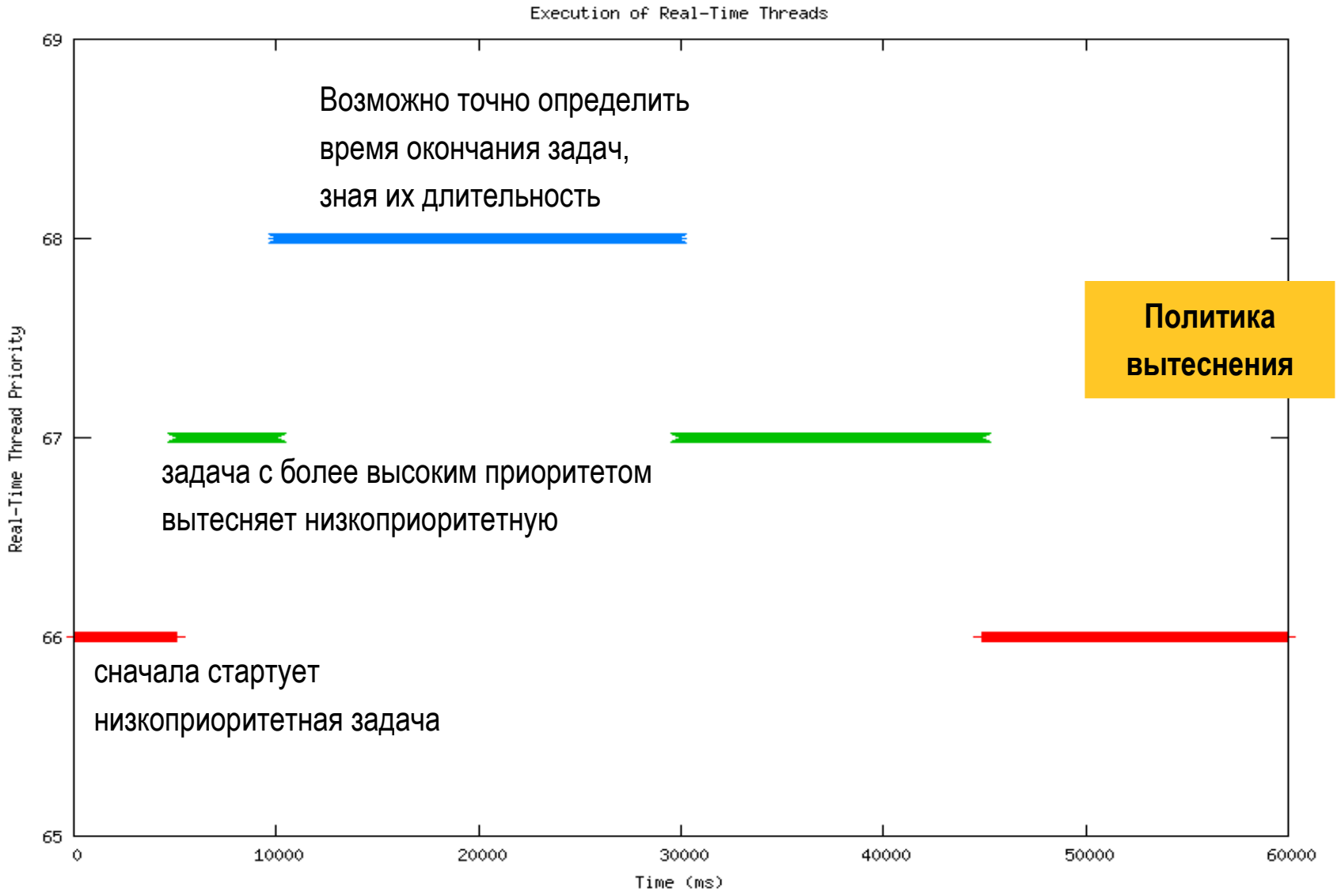
```
Thread t = new java.lang.Thread();
t.setPriority(priority);
```

- на

```
RealtimeThread rtt =
   new javax.realtime.RealtimeThread();
rtt.setSchedulingParameters(prioParams);
```

- И...

Диспетчеризация в Java RTS



Пример диспетчерезации

```
public static void main(String[] args) {
   Thread thread = new Thread() {
   // Thread thread = new RealtimeThread() {
      public void run() {
         System.out.println("RT Thr: started");
         int sum=0;
         for (int j=0; j<=10000; j++)
            for (int i=0; i<=100000; i++) {
               sum += i%10;
         }
         System.out.println("RT Thr: finished, sum=" + sum);
       }
   };
   thread.start();

   System.out.println("NON-RT Thr: before thread.setName");
   thread.setName("RealtimeThread");
   System.out.println("NON-RT Thr: after thread.setName");
}
```

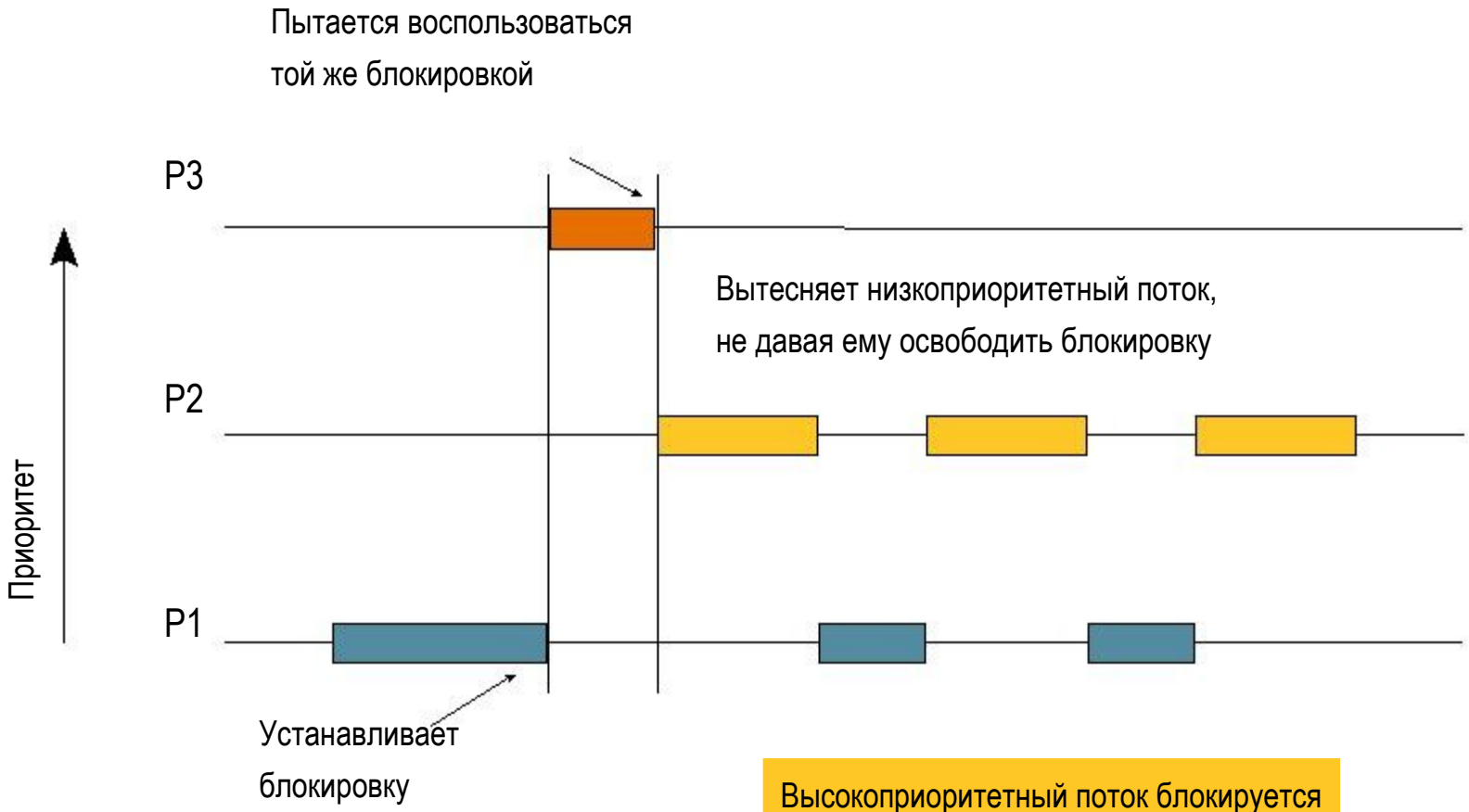
```
NON-RT Thr: before thread.setName
NON-RT Thr: after thread.setName
RT Thr: started
RT Thr: finished, sum=205482704
```

```
RT Thr: started
RT Thr: finished, sum=205482704
NON-RT Thr: before thread.setName
NON-RT Thr: after thread.setName
```

Преимущества

- Платформа RTSJ форсирует выполнение приоритетных потоков
  - Потоки с высоким приоритетом вытесняют потоки с низким приоритетом
  - RTSJ требует как минимум 28 уровней приоритетов
- Политика переключения между потоками строго прописана в RTSJ
  - Исполнение до блокировки, FIFO, вытесняющий диспетчер задач
- Порядок выполнения задач предсказуем разработчику

Инверсия приоритетов



Высокоприоритетный поток блокируется
на неопределённое время

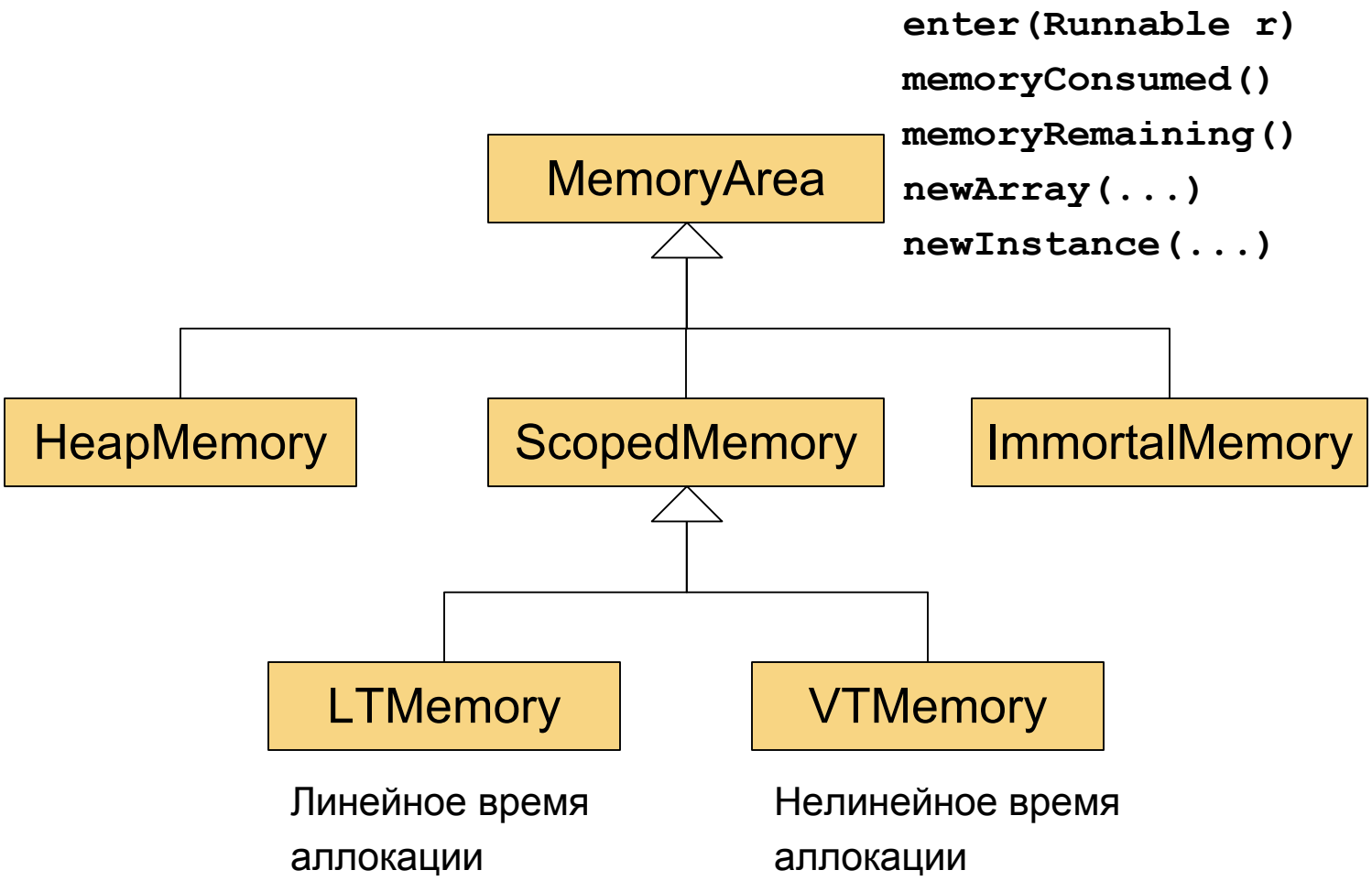
Ликвидация инверсии приоритетов в RTSJ

- Наследование приоритетов (требуется)
  - Приоритет потока, держащего блокировку, временно повышается до приоритета заблокированного потока
  - Не требуется изменение кода
- Priority Ceiling Protocol (не требуется)
  - Каждой блокировке назначается максимальный приоритет потока, который её может взять
  - Потоку, берущему блокировку, назначается этот приоритет до момента её освобождения

Управление памятью

- C/C++: бремя по управлению памятью возлагается на программиста
  - `malloc(), free()`
  - Проблемы: утечки памяти, невалидные указатели
- Java SE: управление памятью автоматическое
  - Вносит неопределённость в поведение приложения, нет контроля над GC
- RTSJ: вводятся области памяти вне heap, в которых не работает GC

Области памяти в RTSJ



Immortal Memory

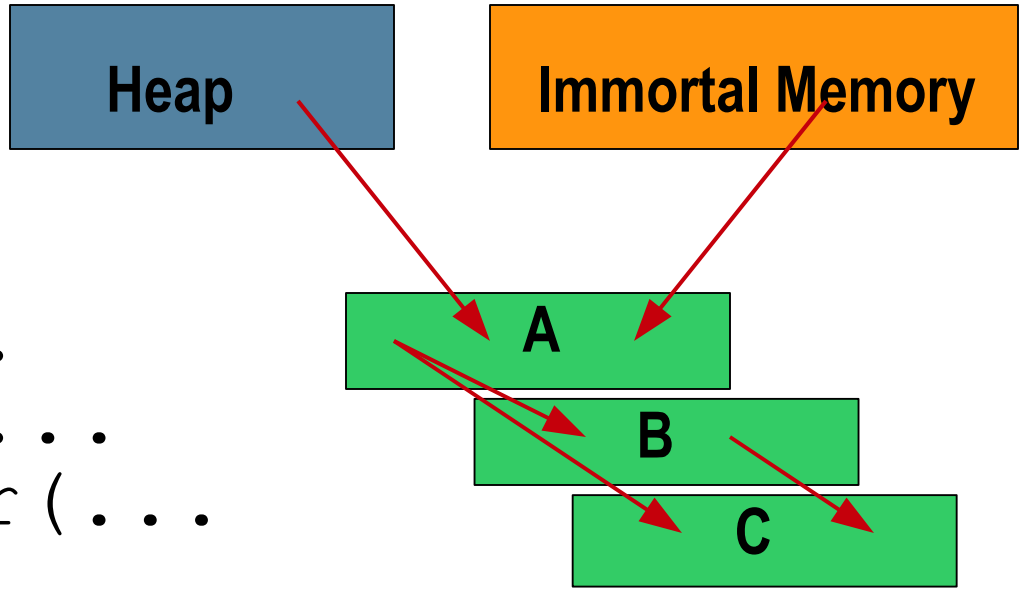
- Разделяется между всеми потоками
- Созданные объекты никогда не удаляются
  - А значит те объекты, на которые они ссылаются, никогда не становятся мусором!
- Создание объектов в Immortal Memory:
  - Неявное
    - static-поля, interned-строки, строковые литералы, объекты классов
  - Явное
    - **`newInstance(), newArray()`**
  - Запуск кода с Immortal Memory в качестве контекста аллокации
    - **`enter(), executeInArea()`**

Scoped Memory

- Время жизни объектов определено областью видимости
  - Объекты существуют, пока область используется потоками
  - Когда потоки выходят из области видимости, scoped memory область может быть освобождена
  - Объекты в Heap/Immortal не могут ссылаться на объекты в Scoped Memory
- В случае нарушения правил присваивания — RuntimeException

Scoped Memory - пример

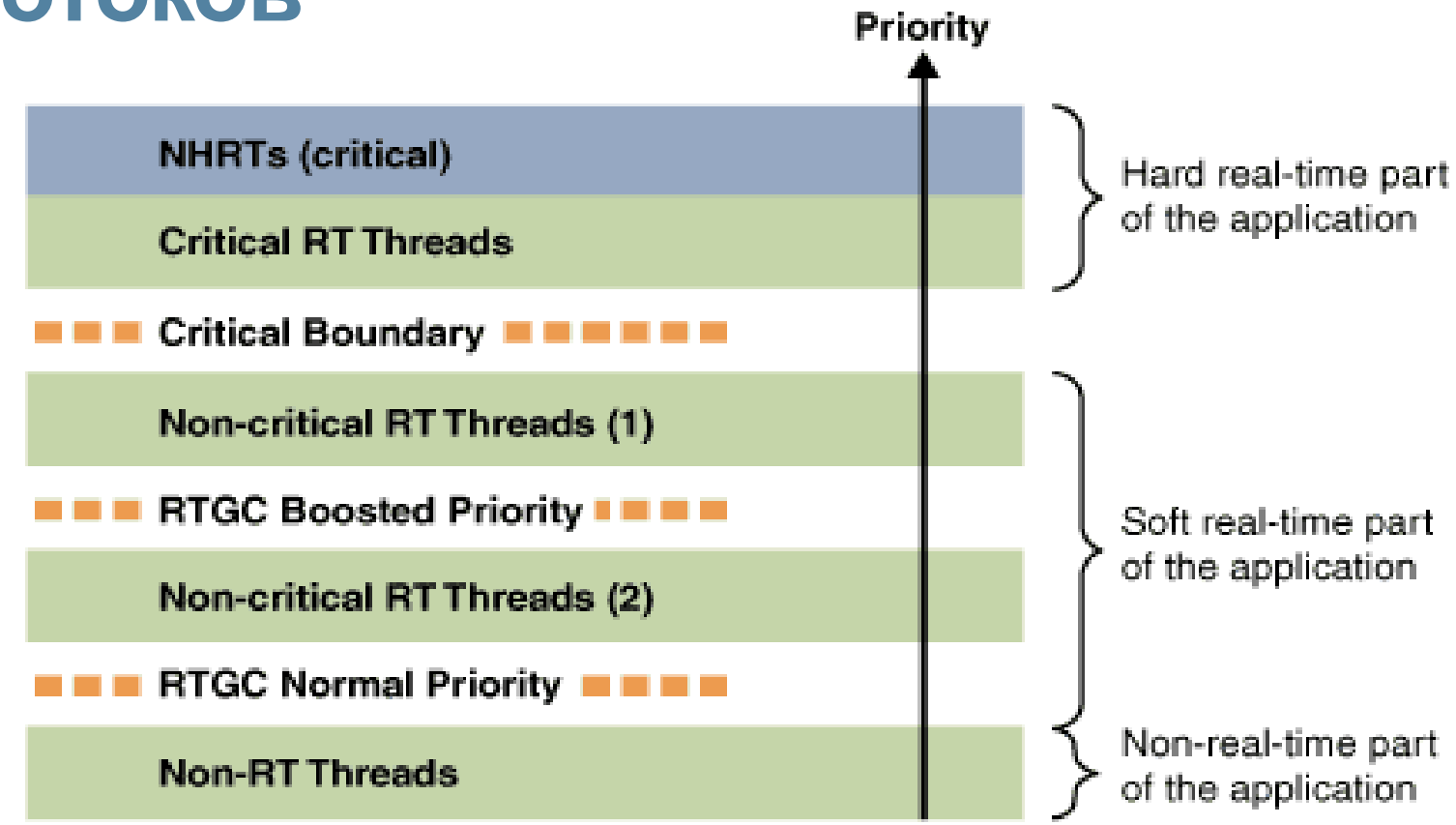
```
A.enter(...
   B.enter(...
      C.enter(...
```



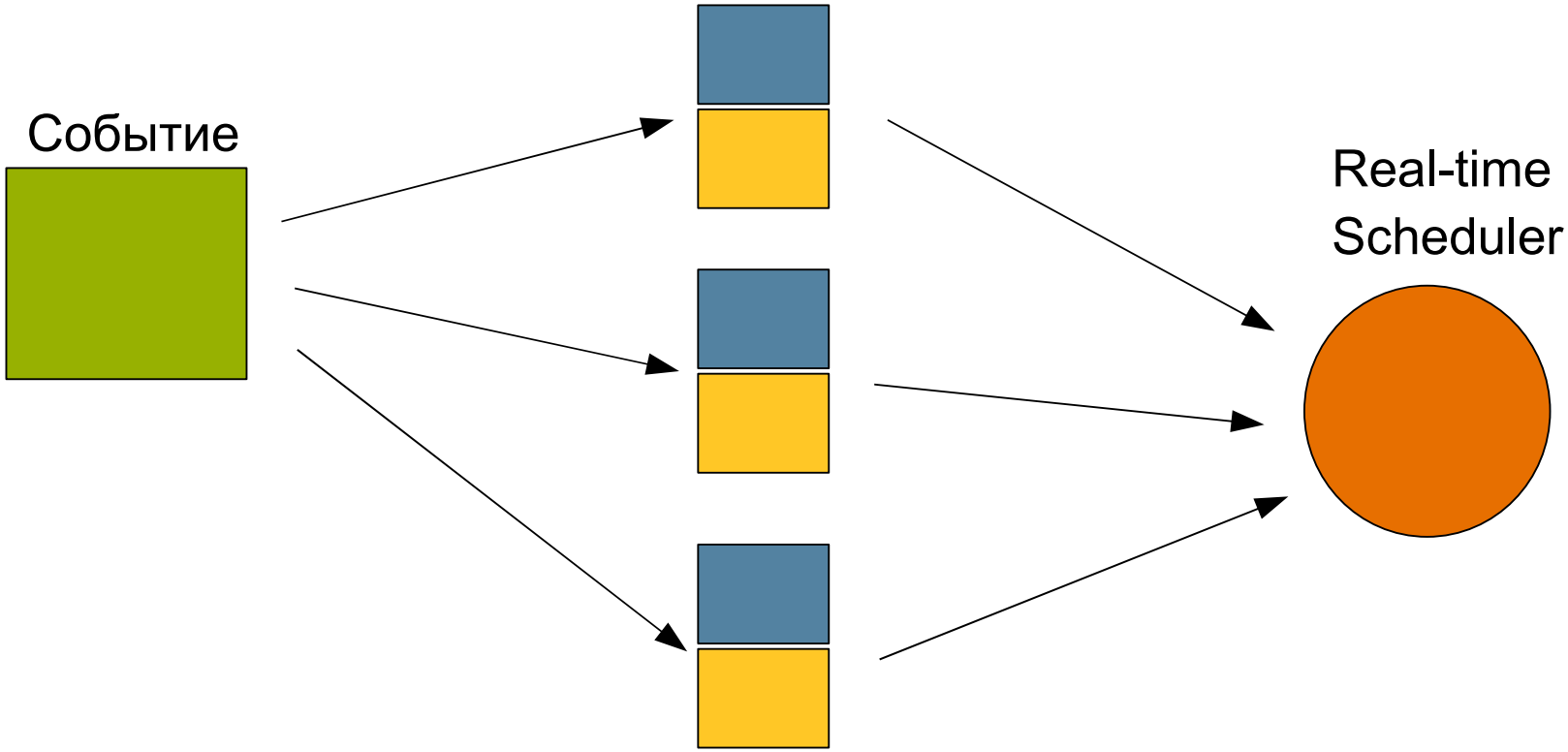
Java RTS: Real-Time Garbage Collector

- RTGC позволяет расширить гарантии малого времени отклика (latency) на RealtimeThread потоки (а не только NoHeapRealtimeThread)
- Критические RealtimeThread'ы могут вытеснять Garbage Collector
  - … и при этом создавать объекты в специальном зарезервированном буфере
  - … и тем самым избегать непредсказуемость времени отклика связанную с GC
- Стоимость выполнения RTGC „оплачивается” некритическими потоками

Java RTS: уровни приоритетов потоков

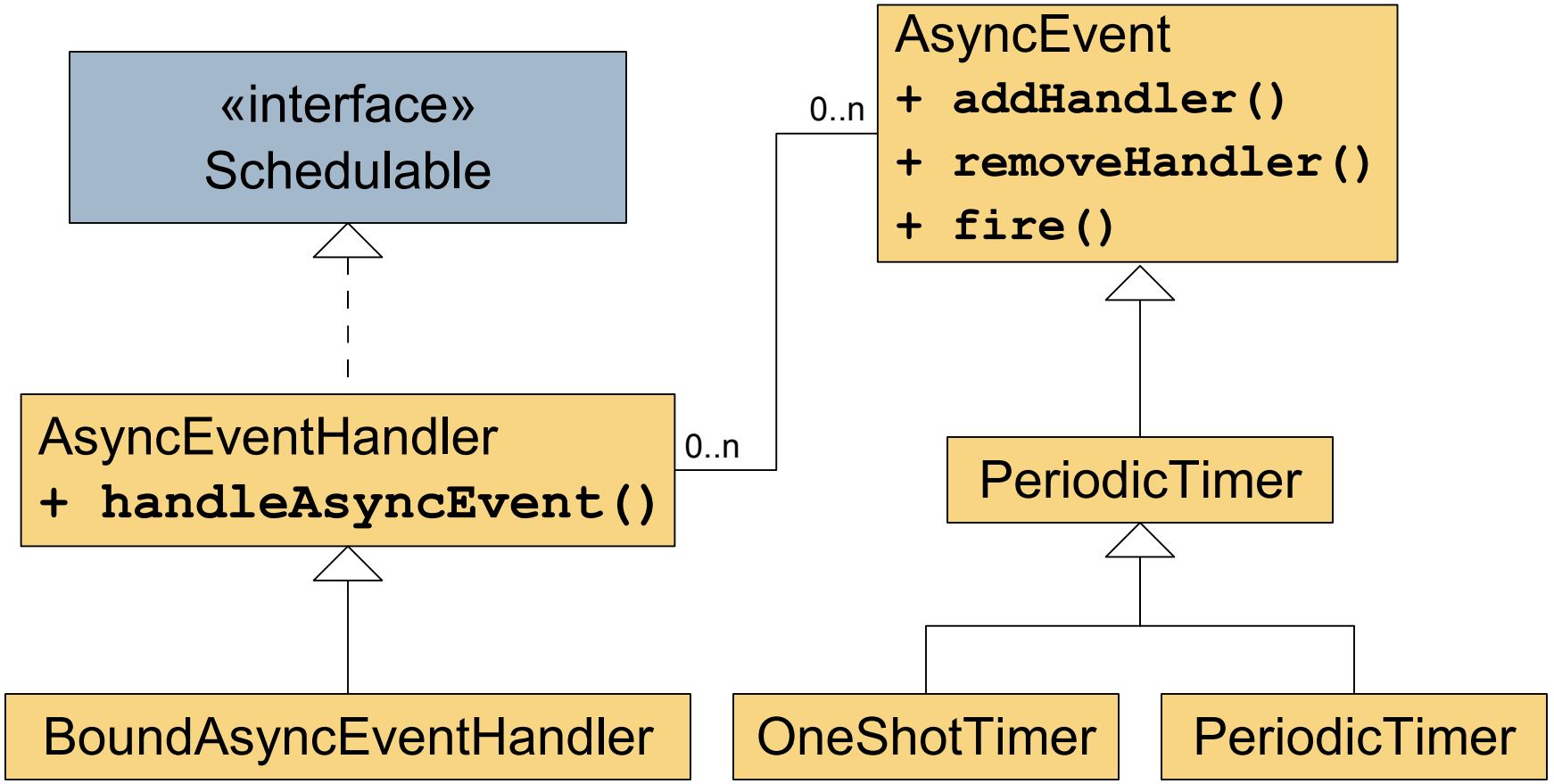


Архитектура асинхронных событий



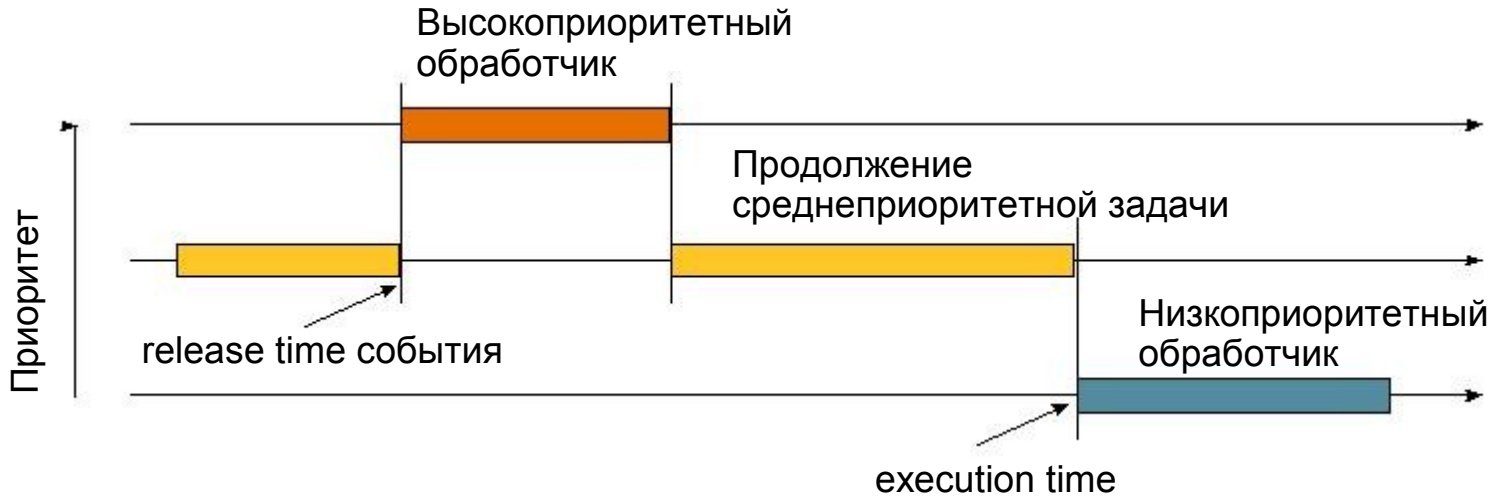
Обработчики событий:
логика + real-time параметры

Асинхронные события и обработчики



Release vs. Execution

- Когда происходит событие
  - Все связанные обработчики запускаются (released)
  - Обработчики начинают выполняться в соответствии со своими real-time параметрами



AsyncEvent и AsyncEventHandler

- Создание события

```
AsyncEvent event = new AsyncEvent();
```

- Логика обработки

```
AsyncEventHandler handler = new AsyncEventHandler() {
    public void handleAsyncEvent() {
        // обрабатываем событие
    }
};
event.addHandler(handler);
```

- Установка параметров диспетчеризации

```
SchedulingParameters sp = new PriorityParameters(
    PriorityScheduler.instance().getMaxPriority());
handler.setSchedulingParameters(sp);
```

- Активация события

```
event.fire();
```

Типы AsyncEventHandler'ов

- AsyncEventHandler
  - Динамически связывается с потоком операционной системы
  - Оптимальное использование ресурсов
- BoundAsyncEventHandler
  - Класс-наследник AsyncEventHandler
  - Постоянная привязка к потоку операционной системы
  - Минимальная задержка (latency)

Периодическое выполнение

- Используется в реализации большинства систем управления
  - Системы с обратной связью
  - ПИД-регуляторы
- Два подхода к реализации в RTSJ:
  - Использование класса PeriodicTimer совместно с набором обработчиков его событий
  - Использование waitForNextPeriod в потоке и установка параметров периодичного разблокирования потока

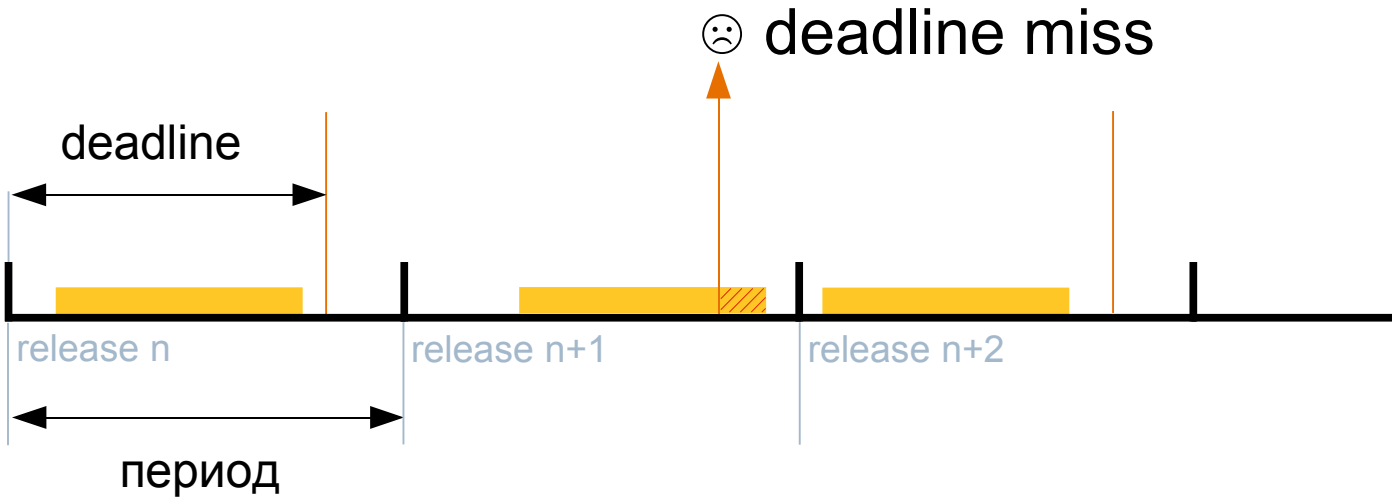
Создание периодически-разблокируемых Real-Time потоков

```
ReleaseParameters rp =
   new PeriodicParameters(start, period);

RealtimeThread thread =
   new RealtimeThread(null, rp) {
      public void run() {
         while (true) {
            … // логика управления
            waitForNextPeriod();
         }
      }
   };
thread.start();
```

Мониторинг выполнения

- RTSJ предоставляет возможности для мониторинга и реагирования на ненормальное поведение



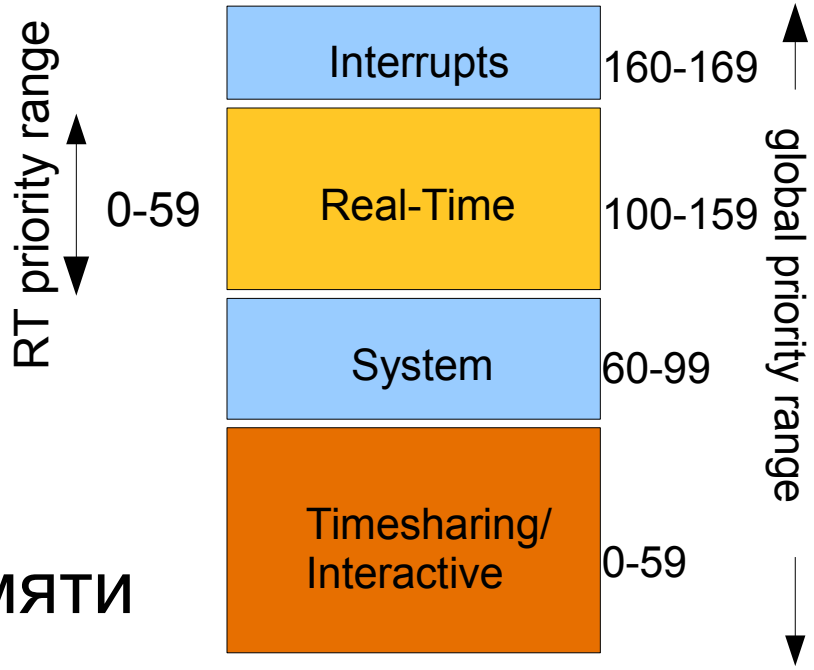
- “deadline miss” событие инициируется в момент, когда задача нарушила deadline

Использование существующих Java-компонент

- Возможно
  - … но надо следить за памятью и O(·)
  - … и за использованием synchronized
- Сложности
  - static-объекты живут в ImmortalMemory
    - а также всё, на что они ссылаются, не освобождается
  - Классы не выгружаются
  - JIT-компиляция должна по возможности заменятся AOT-компиляцией
  - Стандартные библиотеки Java SE плохо подходят для разработки real-time приложений

Java RTS на Solaris

- 60 уровней приоритетов
- Выше только приоритеты прерываний
- Хорошо масштабируется на мультипроцессорные системы
- JVM зафиксирована в памяти



Разделение ресурсов

- Можно назначить конкретные задачи, которыми занимаются процессоры
  - > Processor sets pools, containers
  - > Возможно выделить процессоры под критические задачи
  - > Уменьшаем количество кэш-промахов
- Можно запретить аппаратные прерывания на части процессоров

1 x core for hard RT threads; set to *no-intr*

1 x core for soft RT threads

2 x cores for non-RT threads

Предсказуема ли Java?

Правило трех *P*

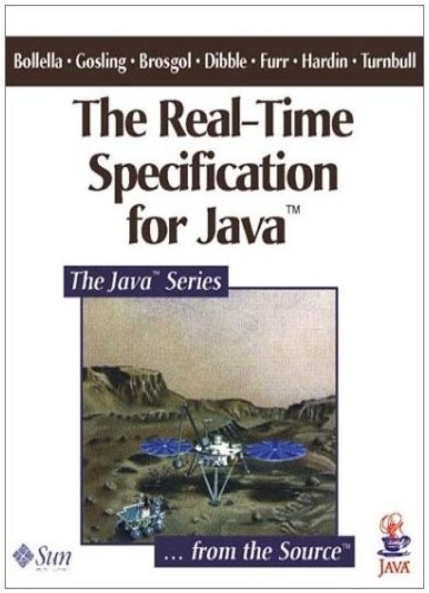
- **P**rioritize
  - приоритезируйте потоки приложения согласно их важности
  - в JRTS поток с большим приоритетом всегда исполняется первым
- **P**artition
  - поделите ресурсы между потоками
  - гарантируйте наличие cpu и памяти для критических потоков
- **P**repare
  - в критических потоках не должно случаться непредсказуемых действий
  - **под**грузите классы, **пре**-инициализируйте классы, **пре**-компилируйте методы

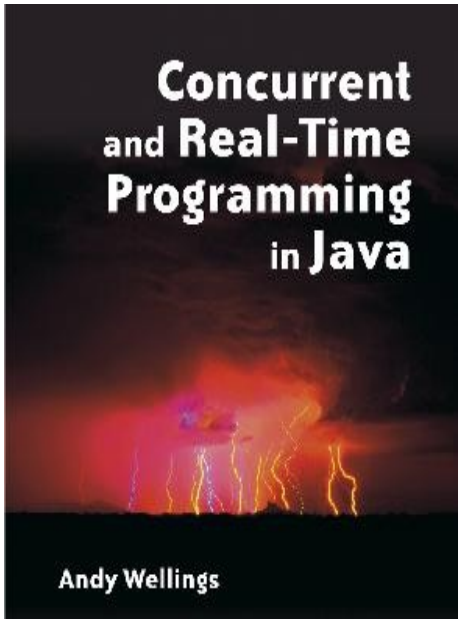
Демонстрация

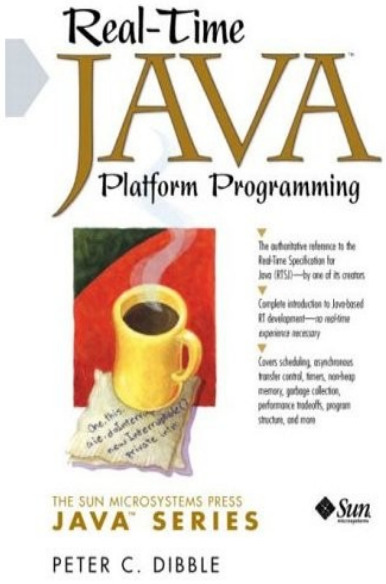
http://www.youtube.com/watch?v=lXct7Bzdhzw

Хотите попробовать?

- Java RTS 2.1ea доступна для скачивания (90 дней evaluation)
  - > http://java.sun.com/javase/technologies/realtime
  - > Официальное поддерживается Solaris, SUSE Linux Enterprise Real Time 10, Red Hat Enterprise MRG 1.0
  - > …но работает и на других Linux-дистрибутивах
- Спецификация: http://www.rtsj.org







Хотите узнать больше? (1)

- Презентации с JavaOne
  - http://java.sun.com/javase/technologies/realtime/reference.jsp#JavaOne
- Блоги разработчиков
  - http://blogs.sun.com/bollellaRT/
  - http://blogs.sun.com/delsart/
  - http://blogs.sun.com/dholmes/
  - http://blogs.sun.com/roland/
  - http://blogs.sun.com/therk
- И многое другое на
  - http://java.sun.com/javase/technologies/realtime/reference.jsp

Хотите узнать больше? (2)

- Форумы и блоги на русском языке
  - http://developers.sun.ru
    спрашивайте про RTJ в разделе Java SE
  - http://blogs.sun.com/vmrobot/category/Real+Time+Java
- Слайды доклада “*Разработка систем реального времени при помощи Sun Java Real-Time System*”, конференция TechDays 2008, Санкт-Петербург
  http://developers.sun.ru/techdays2008

Спасибо!

Sun Microsystems
Ekaterina.Pavlova@Sun.COM