Concurrency in Java and Real-Time Operating Systems

Languages for Embedded Systems

Prof. Stephen A. Edwards

Columbia University

March 2009
The Java Language

Developed by James Gosling et al. at Sun Microsystems in the early 1990s

Originally called “Oak”

First intended application was as an OS for TV set top boxes

Main goals were portability and safety

Originally for embedded consumer software
The Java Language

Set-top boxes: nobody cared at the time

Next big application: “applets,” little programs dynamically added to web browsers

A partial failure despite aggressive Sun marketing.

- Incompatible Java implementations
- Few users had enough bandwidth
- Fantastically slow Java interpreters

Javascript, a high-level scripting language that has nothing to do with the Java language, but leverages Sun’s marketing, has taken over this role.
Aside: The JVM

```java
int gcd(int a, int b) {
    while (a != b) {
        if (a > b)
            a -= b;
        else
            b -= a;
    }
    return a;
}
```

A stack-based machine language. Mostly low-level operations (e.g., add top two stack elements), but also virtual method dispatch.

```
# javap -c Gcd

Method int gcd(int, int)
    0 goto 19

3 iload_1    // Push a
4 iload_2    // Push b
5 if_icmple 15 // if a <= b goto 15

8 iload_1    // Push a
9 iload_2    // Push b
10 isub       // a - b
11 istore_1   // Store new a
12 goto 19

15 iload_2   // Push b
16 iload_1   // Push a
17 isub       // b - a
18 istore_2   // Store new b

19 iload_1   // Push a
20 iload_2   // Push b
21 if_icmpne 3 // if a != b goto 3

24 iload_1   // Push a
25 ireturn    // Return a
```
Aside: The JVM

Advantages:

- Trivial translation of expressions
- Trivial interpreters
- No problems with exhausting registers
- Often compact

Disadvantages:

- Semantic gap between stack operations and modern register machines
- Hard to see what communicates with what
- Difficult representation for optimization
JVM: The Lesson

If you’re going to interpret something, its level of abstraction must be sufficiently higher than assembly to offset the cost of interpretation.

Java bytecode is stack-based and therefore fairly close to assembly. Usual interpreters run at 1/10th the speed of equivalent C.

Just-in-time compilers, which translate Java bytecode to native assembly and cache the results, are the current solution.
The Java Language

Where does Java succeed?

Enterprise programming

- E.g., dynamic web page generation from large corporate databases in banks
- Environment demands simpler language
- Unskilled programmers, unreleased software
- Speed, Space not critical
- Tends to be run on very large servers
- Main objective is reduced development time
The Java Language

Where does Java succeed?

Education

- Well-designed general-purpose programming language
- Spares programmer from many common pitfalls:
  - Uninitialized pointers
  - Memory management
  - Widely known and used, not just a teaching language

Embedded Systems?

- Jury is still out
Part I

Concurrency
Concurrency in Java

Language supports threads
Multiple contexts/program counters running within the same memory space
All objects shared among threads by default
Fundamentally nondeterministic
Language provide synchronization facilities (monitors) to help avoid nondeterminism
Still a difficult paradigm in which to program
Sun’s libraries reputed to still not be thread-safe
Thread Basics

A thread is a separate program counter
... and stack, local variables, etc.
Not an object or a collection of things
Classes, objects, methods, etc. do not belong to a thread
But a thread may hold a lock on an object
Any method may be executed by one or more threads, even simultaneously
Thread Basics

How to create a thread:

```java
class MyThread extends Thread {
    public void run() { // A thread's "main" method
        /* thread body */
    }
}

MyThread mt = new MyThread; /* Create thread */
mt.start(); /* Start thread at run() */
// Returns immediately
```
implements Runnable vs. extends Thread

An alternative:

```java
class MyRunnable implements Runnable {
    public void run() {
        /* thread body */
    }
}
```

```java
Thread t = new Thread(new MyRunnable());
t.start(); /* Starts thread at run() */
// // Returns immediately}
```

Advantage: class implementing Runnable can be derived from some other class (i.e., not just from Thread).

Disadvantage: “this” is not a Thread so, e.g., sleep() must be called with Thread.currentThread().sleep().
The Sleep Method

class Sleeper extends Thread {
    public void run() {
        for (;;) {
            try {
                sleep(1000); // Pause for at least a second
            } 
            catch (InterruptedException e) {
                return; // caused by thread.interrupt()
            }
            System.out.println("tick");
        }
    }
}

Does this print “tick” once a second?
No: the sleep() delay is merely a lower bound, and it’s not clear how much time the rest of the loop takes.
class PrintingClock implements Runnable {
    public void run() {
        for (;;) {
            java.util.Date now = new java.util.Date();
            System.out.println(now.toString());
            try {
                Thread.currentThread().sleep(1000);
            } catch (InterruptedException e) {}
        }
    }
}

public class Clock {
    public static void main(String args[]) {
        Thread t = new Thread(new PrintingClock());
        t.start();
    }
}
What does the clock print?

$ java Clock
Sat Sep 14 13:04:27 EDT 2002
Sat Sep 14 13:04:29 EDT 2002
Sat Sep 14 13:04:30 EDT 2002
Sat Sep 14 13:04:31 EDT 2002

What happened to 13:04:28?
Part II

Synchronization
Motivation for Synchronization

Something you might want to implement:

```java
class ScoreKeeper {
    int _score = 0;
    void score(int v) {
        int tmp = _score;
        tmp += v;
        _score = tmp;
    }
}
```

What could the final score be if two threads simultaneously call score(1) and score(2)?
Non-atomic Operations

Java guarantees 32-bit reads and writes to be atomic.

64-bit operations may not be.

Therefore,

```java
int i;
double d;
```

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i = 10; )</td>
<td>( i = 20; )</td>
</tr>
<tr>
<td>( d = 10.0; )</td>
<td>( d = 20.0; )</td>
</tr>
</tbody>
</table>

\( i \) will contain 10 or 20.

\( d \) may contain 10, 20, or garbage.
Per-Object Locks

Each Java object has a lock that may be owned by at least one thread

A thread waits if it attempts to obtain an already-obtained lock

The lock is a counter: one thread may lock an object more than once
A synchronized statement gets an object’s lock before running its body:

```java
Counter mycount = new Counter;

synchronized (mycount) {
    mycount.count();
}
```

Releases the lock when the body terminates.

Choice of object to lock is by convention.
Synchronized Methods

class AtomicCounter {

    private int _count;

    public synchronized void count() {
        _count++;
    }
}

synchronized attribute equivalent to enclosing body with synchronized (this) statement.

Most common way to achieve object-level atomic operations.

Implementation guarantees at most one thread can run a synchronized method on a particular object at once
Deadlock

\begin{verbatim}
synchronized(Foo) {
    synchronized(Bar) {
        // Asking for trouble
    }
}
synchronized(Bar) {
    synchronized(Foo) {
        // Asking for trouble
    }
}
\end{verbatim}

Rule: always acquire locks in the same order.
Synchronization

Say you want a thread to wait for a condition before proceeding.

An infinite loop may deadlock the system (e.g., if it’s using cooperative multitasking).

```
while (!condition) {}
```

Calling `yield` avoids deadlock, but is inefficient:

```
while (!condition) yield();
```

Scheduler may choose to run this thread often, even though condition has not changed.
Java’s Solution: wait() and notify()

wait() like yield(), but requires other thread to reawaken it

```java
while (!condition) wait();
```

Thread that changes the condition calls notify() to resume the thread.

Programmer responsible for ensuring each wait() has a matching notify().
I often have books delivered to the CS department office.

This operation consists of the following steps:

1. Place the order
2. Retrieve the book from the CS department office
3. Place book on bookshelf

Obviously, there’s a delay between steps 1 and 2.
This “order, retrieve, file” thread is running in me, and it needs to wait for the book to arrive.

I could check the department office every minute, hour, day, etc. to see if the book has come in, but this would be waste of time (but possibly good exercise).

Better approach would be to have the “receiving” process alert me when the book actually arrives.

This is what happens: Alice in the front office sends me email saying I have a book.
A Flawed Implementation

class Mailbox {}

public class BookOrder {
    static Mailbox m = new Mailbox();

    public static void main(String args[]) {
        System.out.println("Ordering book");
        Thread t = new Delivery(m);
        t.start();
        synchronized (m) {
            try {
                m.wait(); // Must own lock on m to wait
            } catch (InterruptedException e) {}
        }
        System.out.println("Book Arrived");
    }
}
A Flawed Implementation

```
class Delivery extends Thread {
    Mailbox m;
    Delivery(Mailbox mm) { m = mm; }
    public void run() {
        try {
            sleep(1000); // one-second delivery
        } catch (InterruptedException e) {} 
        synchronized (m) {
            m.notify(); // Must own lock on m to notify it
        }
    }
}
```

What is wrong with this?
A Flawed Implementation

What happens if the book is delivered before the main thread starts waiting?

Answer: the “notify” instruction does not wake up any threads. Later, the main thread starts to wait and will never be awakened.

As if Alice came to my office to tell me when I was not there.
Harder Problem

Sometimes I have a number of books on order, but Alice only tells me a book has arrived. How do I handle waiting for a number of books?

Last solution assumed a single source of notify(); not true in general.

Two rules:

1. Use notifyAll() when more than one thread may be waiting
2. Wait in a while if you could be notified early, e.g.,

```java
while (!condition)
    wait();
```
The multi-book problem

class Mailbox {
    int book;
    Mailbox() { book = -1; }
    void receive_book(int b) { book = b; }
    int which_book() { return book; }
}

This is not thread-safe: we’ll need to synchronize all access to it.
The multi-book problem

The Delivery class also tells the mailbox which book it got.

class Delivery extends Thread {
    Mailbox m;
    int book;
    Delivery(Mailbox mm, int b) {
        m = mm; book = b;
    }
    public void run() {
        try {
            sleep(1000); }
        catch (InterruptedException e) {} 
        synchronized (m) {
            m.receive_book(book);
            m.notifyAll();
        }
    }
}
The multi-book problem

class BookOrder extends Thread {
    int book;
    Mailbox m;
    BookOrder(Mailbox mm, int b) { m = mm; book = b; }
    public void run() {
        System.out.println("Ordering book "+
                        Integer.toString(book));
        Thread t = new Delivery(m, book);
        t.start();
        synchronized (m) {
            while (m.which_book() != book) {
                try { m.wait(); } 
                catch (InterruptedException e) {} 
            }
        }
        System.out.println("Book "+
                        Integer.toString(book) + " Arrived");
    }
}"
The multi-book problem

Finally, the main routine kicks off two ordering threads.

```java
public class MultiOrder {
    static Mailbox m = new Mailbox();
    public static void main(String args[]) {
        BookOrder t1 = new BookOrder(m, 1);
        BookOrder t2 = new BookOrder(m, 2);
        t1.start();
        t2.start();
    }
}
```

$ java MultiOrder
Ordering book 1
Ordering book 2
Book 1 Arrived
Book 2 Arrived
A Better Solution

Last solution relied on threads to synchronize their own access to the Mailbox. Mailbox should be doing this itself:

class Mailbox {
    int book;
    Mailbox() { book = -1; }

    synchronized void deliver(int b) {
        book = b;
        notifyAll();
    }

    synchronized void waitForBook(int b) {
        while (book != b) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
    }
}
class Delivery extends Thread {
    Mailbox m;
    int book;

    Delivery(Mailbox mm, int b) {
        m = mm; book = b;
    }

    public void run() {
        try {
            sleep(1000);
        } catch (InterruptedException e) {} 
        m.deliver(book);
    }
}
class BookOrder extends Thread {
    int book;
    Mailbox m;

    BookOrder(Mailbox mm, int b) {
        m = mm; book = b;
    }

    public void run() {
        System.out.println("Ordering book " +
            Integer.toString(book));
        Thread t = new Delivery(m, book);
        t.start();
        m.waitForBook(book);
        System.out.println("Book " +
            Integer.toString(book) + " Arrived");
    }
}
A Better Solution

```java
public class MultiOrder2 {
    static Mailbox m = new Mailbox();
    public static void main(String args[]) {
        BookOrder t1 = new BookOrder(m, 1);
        BookOrder t2 = new BookOrder(m, 2);
        t1.start();
        t2.start();
    }
}
```

$ java MultiOrder2
Ordering book 1
Ordering book 2
Book 1 Arrived
Book 2 Arrived
Building a Blocking Buffer

Problem: Build a single-place buffer for Objects that will block on write if the buffer is not empty and on read if the buffer is not full.

```java
interface OnePlace {
    public void write(Object o);
    public Object read();
}
```
class MyOnePlace implements OnePlaceBuf {
    Object o;

    public synchronized // ensure atomic updates
    void write(Object oo) {
        try {
            while (o != null)
                wait(); // Block while buffer is full
        } catch (InterruptedException e) {} 

        o = oo; // Fill the buffer
        notifyAll(); // Awaken any waiting processes
    }
}
Blocking Buffer: Read Method

class MyOnePlace implements OnePlaceBuf {

    Object o;

    public synchronized // ensure atomic updates
    Object read() {

        try {
            while (o == null)
                wait(); // Block while buffer is empty
        } catch (InterruptedException e) {
        }

        Object oo = o; // Get the object
        o = null; // Empty the buffer
        notifyAll(); // Awaken any waiting processes
        return oo;
    }
}
class Writer extends Thread {
    OnePlaceBuf b;
    Writer(OnePlaceBuf bb) { b = bb; }
    public void run() {
        for (int i = 0 ; ; i++) {
            b.write(new Integer(i)); // Will block
        }
    }
}

class Reader extends Thread {
    OnePlaceBuf b;
    Reader(OnePlaceBuf bb) { b = bb; }
    public void run() {
        for (;;) {
            Object o = b.read(); // Will block
            System.out.println(o.toString());
        }
    }
}
public class OnePlace {
    static MyOnePlace b = new MyOnePlace();
    public static void main(String args[]) {
        Reader r = new Reader(b);
        Writer w = new Writer(b);
        r.start();
        w.start();
    }
}
Part III

Thread Priorities
Priorities

Each thread has a priority from 1 to 10 (5 typical)
Scheduler’s job is to keep highest-priority threads running

thread.setPriority(5)
What the Language Specification Says

From *The Java Language Specification*:

Every thread has a priority. When there is competition for processing resources, threads with higher priority are *generally* executed in preference to threads with lower priority. Such preference is *not, however, a guarantee* that the highest priority thread will always be running, and thread *priorities cannot be used* to reliably implement mutual exclusion.

Vague enough for you?
Multiple threads at same priority?

Language definition gives implementer freedom

Calling `yield()` suspends current thread to allow other at same priority to run ... maybe

Solaris implementation runs threads until they stop themselves with `wait()`, `yield()`, etc.

Solaris uses co-operative, application-level threads

Windows implementation timeslices because it uses native threads
Starvation

Java does not demand a fair scheduler

Higher-priority threads can consume all resources, prevent lower-priority threads from running

This is called *starvation*

Timing dependent: function of program, hardware, and Java implementation

How do you know if your program suffers from starvation?
Part IV

Implementing Threads
Many operating systems now distinguish between processes and threads:

**Process**  A thread of control running with its own address space, stack, etc. Uses inter-process communication (e.g., Pipes) with other processes.

**Thread**  A thread of control sharing an address space with another thread, but with its own stack, registers, etc. Communicates through shared memory.
Typical Thread Implementations

“Native threads” Implemented by the operating system

- Scheduled by the OS
- Preemptive
- Context switching costly (jump to OS land)
- Can provide true parallelism on multiprocessors

“Green threads” Implemented by the process

- Scheduled by the process
- Cooperative: control only relinquished voluntarily
- Do not permit true parallelism
- Usually faster
Implementing Java Threads

**Many-to-one:** Java VM in a single process/OS thread

Scheduler implemented directly in the JVM

+ Cheaper context switches (no need to involve the OS)
+ Does not rely on particular OS API
  - Must carefully wrap all OS calls to avoid them blocking the process
  - Can’t call other libraries since they may make blocking OS calls
  - Often difficult to support per-thread system objects, e.g., multiple network connections
Implementing Java Threads

**One-to-one:** Each Java thread mapped gets its own OS thread

+ Can exploit multiprocessors (OS can schedule different threads on different processors)
+ No need to wrap OS calls or other libraries
  - More expensive context switching and thread control. Everything is a system call.
  - Less portable
Java Thread Implementations

**Solaris** Supports Light-Weight Processes (OS threads) and Application Threads (in-process threads).

- Java VM uses Application-level threads by default,
- Application thread scheduler can move threads to other LWPs
- Has $2^{31}$ priority levels.
- Java thread implementation is non-preemptive
Java Thread Implementations

Windows 95 etc. Supports OS threads only
- Java VM uses native threads only
- Has only 7 priority levels for threads
- Java thread implementation is preemptive

Linux Supports kernel-level POSIX threads
- Java VM uses native threads
Disturbing Conclusion

Since it is very easy to write a threaded Java program whose behavior differs depending on the scheduling policy, Java programs are not the exemplars of portability as touted by Sun.

For example, consider

```java
bool waiting = true;
while (waiting) {} // wait for waiting=false
```

This deadlocks under a (non-preemptive) green threads implementation, but it might work fine with (preemptive) native threads.
Part V

Thread Miscellany
Thread-Related Methods

Object.wait()  
Wait indefinitely to be notified

Object.wait(long t)  
Wait at most $t$ milliseconds

Object.wait(long t, int n)  
Wait $t$ plus $n$ nanoseconds

Object.notify()  
Release one \textit{waiting} thread

Object.notifyAll()  
Release all \textit{waiting} threads

Thread.interrupt()  
Break from wait, sleep, etc.

Thread.sleep(long t)  
Sleep for $t$ milliseconds

Thread.yield()  
Pass control to another runnable thread

Thread.join()  
Wait for given thread to terminate

Thread.join(long t)  
Wait for termination or timeout

Thread.setPriority(int)  
Set thread’s scheduling priority

Thread.getPriority()  
Get thread’s priority
Java Thread States

- born
- ready
- blocked
- waiting
- running
- sleeping
- dead

Transitions:
- start()
- I/O completed
- notify()
- yield()
- I/O
- wait()
- timeout
- sleep()
- sleep()
- terminate
Deprecated Thread methods

Before JDK 1.2, Thread had three other methods:

- `Thread.stop()` Terminate a thread and release locks
- `Thread.suspend()` Suspend thread without releasing locks
- `Thread.resume()` Resume a suspended thread

These were terribly unsafe and a recipe for disaster:

- `stop()` would release locks, but could easily leave objects in an inconsistent state. Catching the exception it threw would be possible, but insanely messy.
- `suspend()` did not release locks, so it could cause a deadlock if the thread meant to `resume()` it needed locks before it could proceed.
Common Mistakes

- Forgetting to catch the `InterruptedException` potentially thrown by `wait()`, `sleep()`, and other blocking thread operations.
- Forgetting the `notify()` for a corresponding `wait()` (The “Charlie Brown at the Mailbox” error.)
- Waiting on an object without having its lock (i.e., without having synchronized on the object). Throws an `IllegalMonitorStateException`. 
Part VI

Real-Time Operating Systems (RTOSes)
What is an Operating System?

Provides environment for executing programs:
Process abstraction for multitasking/concurrency: Scheduling
Hardware abstraction layer (device drivers)
Filesystems
Communication

We will focus on concurrency and real-time issues
Do I Need One?

Not always

Simplest approach: cyclic executive

```c
for (;;) {
    /* do part of task 1 */
    /* do part of task 2 */
    /* do part of task 3 */
}
```
Cyclic Executive

**Advantages**
- Simple implementation
- Low overhead
- Very predictable

**Disadvantages**
- Can’t handle sporadic events
- Everything must operate in lockstep
- Code must be scheduled manually
Interrupts

Some events can’t wait for next loop iteration:

- Communication channels
- Transient events

Interrupt: environmental event that demands attention

- Example: “byte arrived” interrupt on serial channel

Interrupt routine code executed in response to an interrupt

A solution: Cyclic executive plus interrupt routines
Handling an Interrupt

1. Program runs normally
2. Interrupt occurs
3. Processor state saved
   4. Interrupt routine runs
   5. “Return from Interrupt” instruction runs
4. Processor state restored
5. Normal program execution resumes
Interrupt Service Routines

Most interrupt routines do as little as possible

- Copy peripheral data into a buffer
- Indicate to other code that data has arrived
- Acknowledge the interrupt (tell hardware)

Additional processing usually deferred to outside

E.g., Interrupt causes a process to start or resume running

Objective: let the OS handle scheduling, not the interrupting peripherals
Cyclic Executive Plus Interrupts

Works fine for many signal processing applications
56001 has direct hardware support for this style
Insanely cheap, predictable interrupt handler:

  When interrupt occurs, execute a single user-specified instruction

This typically copies peripheral data into a circular buffer
No context switch, no environment save, no delay
Drawbacks of CE + Interrupts

Main loop still runs in lockstep
Programmer responsible for scheduling
Scheduling static
Sporadic events handled slowly
Cooperative Multitasking

A cheap alternative
Non-preemptive
Processes responsible for relinquishing control
Examples: Original Windows, Macintosh

A process had to periodically call get_next_event() to let other processes proceed

Drawbacks:
  
  Programmer had to ensure this was called frequently
  An errant program would lock up the whole system

Alternative: preemptive multitasking
Concurrency Provided by OS

Basic philosophy:

Let the operating system handle scheduling, and let the programmer handle function

Scheduling and function usually orthogonal

Changing the algorithm would require a change in scheduling

First, a little history
Batch Operating Systems

Original computers ran in batch mode:

Submit job & its input

Job runs to completion

Collect output

Submit next job

Processor cycles very expensive at the time

Jobs involved reading, writing data to/from tapes

Costly cycles were being spent waiting for the tape!
Timesharing Operating Systems

Way to spend time while waiting for I/O:
Let another process run

- Store multiple batch jobs in memory at once
- When one is waiting for the tape, run the other one

Basic idea of timesharing systems

Fairness primary goal of timesharing schedulers
- Let no one process consume all the resources
- Make sure every process gets equal running time
Aside: Modern Computer Architectures

Memory latency now becoming an I/O-like time-waster.

CPU speeds now greatly outstrip memory systems.

All big processes use elaborate multi-level caches.

**An Alternative:**

Certain high-end chips (e.g., Intel’s Xeon) now contain two or three contexts. Can switch among them “instantly.”

Idea: while one process blocks on memory, run another.
Part VII

Real-Time Concurrency
Real-Time Is Not Fair

Main goal of an RTOS scheduler: meeting deadlines

If you have five homework assignments and only one is due in an hour, you work on that one

Fairness does not help you meet deadlines
Priority-based Scheduling

Typical RTOS has on fixed-priority preemptive scheduler

Assign each process a priority

At any time, scheduler runs highest priority process ready to run (processes can be blocked waiting for resources).

Process runs to completion unless preempted
Typical RTOS Task Model

Each task a triplet: (execution time, period, deadline)

Usually, deadline = period

Can be initiated any time during the period

$p = (2, 8, 8)$
Example: Fly-by-wire Avionics

Hard real-time system with multirate behavior

- **gyros/accel**: INU 1 kHz → Pitch ctrl. 500 Hz → Aileron 1 1 kHz → Aileron
- **GPS**: GPS 20 Hz → Lateral ctrl. 250 Hz → Aileron 2 1 kHz → Aileron
- **Sensor**: Air data 1 kHz → Throttle ctrl. 250 Hz → Elevator 1 kHz → Elevator
- **Stick**: Joystick 500 Hz → Rudder 1 kHz → Rudder

Air data 1 kHz

Throttle ctrl. 250 Hz
Priority-based Preemptive Scheduling

Always run the highest-priority runnable process.

```
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Solutions to equal priorities

- Simply prohibit: Each process has unique priority
- Time-slice processes at the same priority
  - Extra context-switch overhead
  - No starvation dangers at that level
- Processes at the same priority never preempt
  - More efficient
  - Still meets deadlines if possible
Part VIII

Rate-Monotonic Scheduling
Rate-Monotonic Scheduling

Common way to assign priorities

Result from Liu & Layland, 1973 (JACM)

Simple to understand and implement:

*Processes with shorter period given higher priority*

E.g.,

<table>
<thead>
<tr>
<th>Period</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1 (high)</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>4 (low)</td>
</tr>
</tbody>
</table>
Rate-monotonic scheduling is optimal:

If there is fixed-priority schedule that meets all deadlines, then RMS will produce a feasible schedule

Task sets do not always have a schedule

Simple example: P1 = (10, 20, 20) P2 = (5, 9, 9)

Requires more than 100% processor utilization
RMS Missing a Deadline

\[ p_1 = (2, 4, 4), \quad p_2 = (3, 6, 6), \quad 100\% \text{ utilization} \]

\[ \begin{array}{c}
1 \\
2 \\
1 \\
2 \\
1 \\
2 \\
1 \\
2 \\
\end{array} \]

\[ p_2 \text{ misses a deadline} \]

Changing \( p_2 = (2, 6, 6) \) would have met the deadline and reduced utilization to 83\%. 
When Is There an RMS Schedule?

Key metric is processor utilization: sum of compute time divided by period for each process:

\[ U = \sum_i \frac{c_i}{p_i} \]

No schedule can possibly exist if \( U > 1 \) No processor can be running 110% of the time

Fundamental result: RMS schedule exists if

\[ U < n(2^{1/n} - 1) \]

Proof based on case analysis (P1 finishes before P2)
<table>
<thead>
<tr>
<th>$n$</th>
<th><strong>Bound for $U$</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100% Trivial: one process</td>
</tr>
<tr>
<td>2</td>
<td>83% Two process case</td>
</tr>
<tr>
<td>3</td>
<td>78%</td>
</tr>
<tr>
<td>4</td>
<td>76%</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$\infty$</td>
<td>69% Asymptotic bound</td>
</tr>
</tbody>
</table>
When Is There an RMS Schedule?

Asymptotic result:

If the required processor utilization is under 69%, RMS will give a valid schedule

Converse is not true. Instead:

If the required processor utilization is over 69%, RMS might still give a valid schedule, but there is no guarantee
EDF Scheduling

RMS assumes fixed priorities.
Can you do better with dynamically-chosen priorities?
Earliest deadline first:

*Processes with soonest deadline given highest priority*
EDF Meeting a Deadline

\[ p_1 = (2, 4, 4), \ p_2 = (3, 6, 6), \] 100% utilization

takes priority with its earlier deadline
Key EDF Result

Earliest deadline first scheduling is optimal:

If a dynamic priority schedule exists, EDF will produce a feasible schedule

Earliest deadline first scheduling is efficient:

A dynamic priority schedule exists if and only if utilization is no greater than 100%
RMA only guarantees feasibility at 69% utilization, EDF guarantees it at 100%.

EDF is complicated enough to have unacceptable overhead.

More complicated than RMA: harder to analyze.

Less predictable: can’t guarantee which process runs when.
Part IX

Priority Inversion
Priority Inversion

RMS and EDF assume no process interaction, often a gross oversimplification

Process 1 misses deadline
Process 1 blocked waiting for resource
Process 1 preempts Process 2
Process 2 acquires lock on resource
Process 2 begins running
Priority Inversion

Lower-priority process effectively blocks a higher-priority one
Lower-priority process's ownership of lock prevents higher-priority process from running
Nasty: makes high-priority process runtime unpredictable
Nastier Example

Process 2 blocks Process 1 indefinitely

Process 2 delays Process 3
Process 1 blocked, needs lock from Process 3
Process 1 preempts Process 2
Process 2 preempts Process 3
Process 3 acquires lock on resource
Process 3 begins running
Priority Inheritance

Solution to priority inversion

Increase process’s priority while it possesses a lock

Level to increase: highest priority of any process that might want to acquire same lock

I.e., high enough to prevent it from being preempted

Danger: Low-priority process acquires lock, gets high priority and hogs the processor

So much for RMS
Priority Inheritance

Basic rule: low-priority processes should acquire high-priority locks only briefly

An example of why concurrent systems are so hard to analyze

RMS gives a strong result

No equivalent result when locks and priority inheritance is used
Summary

Cyclic executive—A way to avoid an RTOS

Adding interrupts helps somewhat

Interrupt handlers gather data, acknowledge interrupt as quickly as possible

Cooperative multitasking, but programs don’t like to cooperate
Summary

Preemptive Priority-Based Multitasking—Deadlines, not fairness, the goal of RTOSes

Rate-monotonic analysis

- Shorter periods get higher priorities
- Guaranteed at 69% utilization, may work higher

Earliest deadline first scheduling

- Dynamic priority scheme
- Optimal, guaranteed when utilization 100% or less
Summary

Priority Inversion

- Low-priority process acquires lock, blocks higher-priority process
- Priority inheritance temporarily raises process priority
- Difficult to analyze