11. Vícevláknové programování

Obsah

- Základní pojmy (vlákno, proces, monitor)
- Synchronizace
- Neměnné objekty (immutable)
- Zablokování (deadlock)
- Stavy a plánování vláken
- Metody synchronizace
- Metody komunikace mezi vlákny
- Vhodné užití a příklady

Základní pojmy

Proces

- **Proces** je instance běžícího programu
- Má vlastní paměťový prostor oddělený od ostatních procesů
- Obsahuje kód programu, data, heap, zásobník a další systémové zdroje
- Procesy jsou navzájem izolované - jeden proces nemůže přímo přistupovat k paměti jiného procesu
- Komunikace mezi procesy vyžaduje použití speciálních mechanismů (IPC - Inter-Process Communication)

Vlákno

- **Vlákno** (thread) je nejmenší jednotka zpracování, kterou může operační systém naplánovat k vykonání
- Vlákna existují v rámci procesu a sdílejí jeho paměťový prostor
- Každé vlákno má vlastní:
  - Programový čítač (PC)
  - Registry
  - Zásobník
  - Stav
- Vlákna ve stejném procesu sdílejí:
  - Kód programu
  - Data
  - Heap
  - Systémové zdroje (otevřené soubory, síťová spojení)

Monitor

- **Monitor** je synchronizační mechanismus, který umožňuje vláknům vzájemně výlučný přístup ke sdíleným zdrojům
- V Javě je monitor implementován pomocí klíčového slova `synchronized`
- Každý objekt v Javě má asociovaný zámek (lock), který se používá při synchronizaci
- Když vlákno vstoupí do synchronizovaného bloku, získá zámek objektu a drží ho, dokud blok neopustí

Synchronizace

Synchronizace je proces koordinace vláken tak, aby předešla problémům při současném přístupu ke sdíleným zdrojům.

Problémy bez synchronizace

1. **Race Condition** - výsledek operace závisí na pořadí a načasování, ve kterém vlákna přistupují ke sdíleným datům
2. **Ztráta aktualizace** - jedno vlákno přepíše změny provedené jiným vláknem

3. **Nekonzistentní čtení** - vlákno čte částečně aktualizovaná data

Synchronizační techniky v Javě

1. Blok synchronizace

```
synchronized (objekt) {
    // Kritická sekce - pouze jedno vlákno může být v tomto bloku pro daný objekt
}
```

2. Synchronizované metody

```
public synchronized void metoda() {
    // Celá metoda je kritická sekce
}
```

3. Třídy pro atomické operace (java.util.concurrent.atomic)

```
import java.util.concurrent.atomic.AtomicInteger;

AtomicInteger counter = new AtomicInteger(0);
counter.incrementAndGet(); // Atomická operace zvýšení o 1
```

4. Zámky (java.util.concurrent.locks)

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

Lock lock = new ReentrantLock();
lock.lock();
try {
    // Kritická sekce
} finally {
    lock.unlock(); // Vždy odemknout, i v případě výjimky
}
```

Neměnné objekty (immutable)

Neměnný objekt je objekt, jehož stav nelze změnit po jeho vytvoření.

Charakteristiky neměnných objektů

- Všechny atributy jsou finální (final)
- Třída může být označena jako finální, aby ji nebylo možné rozšířit
- Žádné metody, které by měnily stav objektu
- Všechny proměnné objektu jsou také neměnné nebo je zajištěno, že jejich stav zůstane nezměněn

Příklad neměnného objektu v Javě

```
public final class ImmutablePerson {
    private final String name;
    private final int age;

    public ImmutablePerson(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
```

```
        return name;
    }

    public int getAge() {
        return age;
    }

    // Neexistují žádné settery nebo metody měnící stav
}
```

Význam neměnných objektů ve vícevláknovém programování

1. **Vláknová bezpečnost** - neměnné objekty jsou přirozeně vláknově bezpečné, protože jejich stav nelze změnit
2. **Eliminace race condition** - není potřeba synchronizace při čtení
3. **Sdílení bez rizika** - objekty mohou být bezpečně sdíleny mezi vlákny
4. **Předvídatelnost** - stav objektu je vždy stejný, což usnadňuje ladění
5. **Cachování a optimalizace** - neměnné objekty mohou být bezpečně uloženy v cache

Příklady neměnných tříd v Javě

- `java.lang.String`
- `java.lang.Integer` a další wrappery primitivních typů
- `java.math.BigInteger` a `java.math.BigDecimal`
- `java.time.LocalDate` a další třídy v balíčku Date-Time API

Zablokování (deadlock)

Deadlock je situace, kdy dvě nebo více vláken čekají na uvolnění zdrojů, které drží jiná vlákna, a kvůli tomu žádné z nich nemůže pokračovat.

Podmínky pro vznik deadlocku (všechny musí být splněny současně)

1. **Vzájemné vyloučení** - zdroje nemohou být sdíleny (v daný okamžik může zdroj používat pouze jedno vlákno)
2. **Držení a čekání** - vlákno drží nějaké zdroje a zároveň čeká na další
3. **Nepreemptivnost** - zdroje nelze odebrat vláknu, které je drží (vlákno je musí samo uvolnit)
4. **Cyklické čekání** - existuje cyklus vláken, kde každé vlákno čeká na zdroj držený následujícím vláknem v cyklu

Příklad deadlocku v Javě

```
public class DeadlockExample {
    private final Object resource1 = new Object();
    private final Object resource2 = new Object();

    public void method1() {
        synchronized (resource1) {
            System.out.println("Vlákno 1: Držím resource1");

            try {
                Thread.sleep(100); // Zvýší pravděpodobnost deadlocku
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println("Vlákno 1: Čekám na resource2");
            synchronized (resource2) {
```

```
                System.out.println("Vlákno 1: Držím resource1 a resource2");
            }
        }
    }

    public void method2() {
        synchronized (resource2) {
            System.out.println("Vlákno 2: Držím resource2");

            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println("Vlákno 2: Čekám na resource1");
            synchronized (resource1) {
                System.out.println("Vlákno 2: Držím resource1 a resource2");
            }
        }
    }

    public static void main(String[] args) {
        DeadlockExample deadlock = new DeadlockExample();

        new Thread(() -> {
            deadlock.method1();
        }).start();

        new Thread(() -> {
            deadlock.method2();
        }).start();
    }
}
```

Prevence deadlocku

1. **Prevence cyklického čekání** - zavést pevné pořadí získávání zámků

```
// Správný způsob - vždy získávat zámky ve stejném pořadí
public void methodSafe() {
    synchronized (resource1) {
        synchronized (resource2) {
            // Kritická sekce
        }
    }
}
```

2. **Použití tryLock** - získat zámek s časovým limitem

```
Lock lock1 = new ReentrantLock();
Lock lock2 = new ReentrantLock();

boolean getLocks() {
    try {
        boolean gotFirstLock = lock1.tryLock(1, TimeUnit.SECONDS);
        if (gotFirstLock) {
            try {
                boolean gotSecondLock = lock2.tryLock(1, TimeUnit.SECONDS);
                if (gotSecondLock) {
```

```
                    return true; // Máme oba zámky
                }
            } finally {
                lock1.unlock(); // Uvolnit první zámek, pokud druhý nezískáme
            }
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
    return false; // Nezískali jsme oba zámky
}
```

3. **Detekce deadlocku** - Java poskytuje nástroje pro detekci deadlocku (JConsole, JVisualVM)
4. **Plánování vláken** - nastavení timeout pro operace získávání zdrojů

Stavy a plánování vláken

Stavy vlákna v Javě

Vlákno může být v jednom z těchto stavů:

1. **NEW** - vlákno bylo vytvořeno, ale ještě nebylo spuštěno (metoda `start()` nebyla zavolána)
2. **RUNNABLE** - vlákno je připraveno k běhu nebo běží (po zavolání `start()`)
3. **BLOCKED** - vlákno je blokováno a čeká na monitor lock
4. **WAITING** - vlákno čeká, dokud ho jiné vlákno neprobudí (`wait()`, `join()`)
5. **TIMED_WAITING** - vlákno čeká určitý čas (`sleep()`, `wait(timeout)`, `join(timeout)`)
6. **TERMINATED** - vlákno dokončilo svou činnost

Přechody mezi stavy

- **NEW** → **RUNNABLE**: po zavolání `start()`
- **RUNNABLE** → **BLOCKED**: při pokusu o získání monitoru, který drží jiné vlákno
- **RUNNABLE** → **WAITING**: po zavolání `wait()`, `join()` nebo `park()`
- **RUNNABLE** → **TIMED_WAITING**: po zavolání `sleep()`, `wait(timeout)`, `join(timeout)`, nebo `parkNanos()`, `parkUntil()`
- **BLOCKED** → **RUNNABLE**: když vlákno získá monitor, na který čekalo
- **WAITING** → **RUNNABLE**: po zavolání `notify()`/`notifyAll()` na objektu, na kterém bylo zavoláno `wait()`, nebo po zavolání `unpark()`
- **TIMED_WAITING** → **RUNNABLE**: po vypršení času nebo po zavolání `notify()`/`notifyAll()`/`unpark()`
- Jakýkoliv stav → **TERMINATED**: po dokončení běhu nebo při neošetřené výjimce

Plánování vláken

- V Javě plánování vláken provádí JVM ve spolupráci s operačním systémem
- Javovské vlákna jsou typicky mapována na nativní vlákna OS
- Priority vláken:
  - `Thread.MIN_PRIORITY` (1)
  - `Thread.NORM_PRIORITY` (5) - výchozí
  - `Thread.MAX_PRIORITY` (10)
- Nastavení priority: `thread.setPriority(int priority)`
- Vyšší priorita pouze navrhuje JVM, že by dané vlákno mělo dostat více procesorového času, ale nezaručuje to

Metody synchronizace

1. Intrinsic Locks (synchronized)

```
// Synchronizace na objektu
synchronized (object) {
```

```
    // Kritická sekce
}

// Synchronizace metody - implicitní zámek je 'this'
public synchronized void method() {
    // Kritická sekce
}

// Synchronizovaná statická metoda - zámek je na třídě
public static synchronized void staticMethod() {
    // Kritická sekce
}
```

2. Explicitní zámky (Locks API)

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

Lock lock = new ReentrantLock();
lock.lock();
try {
    // Kritická sekce
} finally {
    lock.unlock();
}
```

Typy explicitních zámků: - **ReentrantLock** - standardní zámek, který může být opakovaně získán stejným vláknem - **ReadWriteLock** - oddělené zámky pro čtení a zápis, umožňuje více vláknům číst současně - **StampedLock** - vylepšená verze ReadWriteLock s optimistickým čtením

3. Atomické operace (Atomic variables)

```
import java.util.concurrent.atomic.AtomicInteger;

AtomicInteger counter = new AtomicInteger(0);
counter.incrementAndGet(); // Atomicky zvýší hodnotu o 1 a vrátí novou hodnotu
counter.getAndIncrement(); // Atomicky vrátí aktuální hodnotu a zvýší o 1
counter.compareAndSet(5, 10); // Atomicky nastaví na 10, pokud je aktuální hodnota 5
```

Dostupné atomické třídy: - **AtomicBoolean**, **AtomicInteger**, **AtomicLong** - **AtomicReference**, **AtomicIntegerArray**, **AtomicLongArray**, **AtomicReferenceArray** - **AtomicIntegerFieldUpdater**, **AtomicLongFieldUpdater**, **AtomicReferenceFieldUpdater**

4. Volatilní proměnné

```
private volatile boolean flag = false;

// Zajišťuje viditelnost změn mezi vlákny, ale nezajišťuje atomicitu operací
```

Vlastnosti `volatile`: - Zajišťuje, že čtení/zápis do proměnné jde vždy do hlavní paměti, ne jen do cache CPU - Zabraňuje přeuspořádání instrukcí překladačem nebo CPU - Nezajišťuje atomicitu složených operací (např. `i++`)

5. Countdown Latch

```
import java.util.concurrent.CountDownLatch;

CountDownLatch latch = new CountDownLatch(3); // Čeká na 3 signály

// Ve vláknu, které čeká
```

```
latch.await(); // Čeká, dokud počítadlo nedosáhne nuly

// Ve vláknech, která signalizují
latch.countDown(); // Sníží počítadlo o 1
```

6. CyclicBarrier

```
import java.util.concurrent.CyclicBarrier;

CyclicBarrier barrier = new CyclicBarrier(3, () -> {
    // Kód, který se provede, když všechna vlákna dosáhnou bariéry
});

// Ve vláknech
barrier.await(); // Čeká, dokud všechna vlákna nedosáhnou bariéry
```

7. Semaphore

```
import java.util.concurrent.Semaphore;

Semaphore semaphore = new Semaphore(3); // Maximálně 3 vlákna současně

semaphore.acquire(); // Získá povolení (sníží počítadlo)
try {
    // Kritická sekce
} finally {
    semaphore.release(); // Uvolní povolení (zvýší počítadlo)
}
```

8. Phaser

```
import java.util.concurrent.Phaser;

Phaser phaser = new Phaser(3); // Registruje 3 účastníky

// Ve vláknech
phaser.arriveAndAwaitAdvance(); // Čeká, dokud všichni účastníci nedorazí
```

Metody komunikace mezi vlákny

1. Sdílená paměť

- Vlákna sdílejí proměnné v heap
- Při sdílení je nutná synchronizace
- Příklad: producent-konzument s využitím sdíleného bufferu

2. Wait-Notify mechanismus

```
// Vlákno čekající na signál
synchronized (sharedObject) {
    while (!condition) {
        sharedObject.wait(); // Uvolní zámek a uspí vlákno
    }
    // Pokračuje po signálu a ověření podmínky
}

// Vlákno signalizující
synchronized (sharedObject) {
    // Změna stavu
```

```
    condition = true;
    sharedObject.notify(); // Probudí jedno čekající vlákno
    // NEBO
    sharedObject.notifyAll(); // Probudí všechna čekající vlákna
}
```

3. Podmínkové proměnné (Condition)

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

Lock lock = new ReentrantLock();
Condition condition = lock.newCondition();

// Vlákno čekající na signál
lock.lock();
try {
    while (!conditionMet) {
        condition.await(); // Uvolní zámek a uspí vlákno
    }
    // Pokračuje po signálu
} finally {
    lock.unlock();
}

// Vlákno signalizující
lock.lock();
try {
    // Změna stavu
    conditionMet = true;
    condition.signal(); // Probudí jedno čekající vlákno
    // NEBO
    condition.signalAll(); // Probudí všechna čekající vlákna
} finally {
    lock.unlock();
}
```

4. Blocking Queue

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

BlockingQueue<Task> queue = new LinkedBlockingQueue<>(10); // Kapacita 10

// Producent
queue.put(new Task()); // Blokuje, pokud je fronta plná

// Konzument
Task task = queue.take(); // Blokuje, pokud je fronta prázdná
```

Typy blokujících front: - **LinkedBlockingQueue** - založená na spojeném seznamu, volitelná kapacita - **ArrayBlockingQueue** - založená na poli, pevná kapacita - **PriorityBlockingQueue** - prioritní fronta - **DelayQueue** - prvky jsou dostupné až po vypršení jejich zpoždění - **SynchronousQueue** - fronta s kapacitou 0, předání probíhá přímo mezi vlákny

5. Future a CompletableFuture

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

ExecutorService executor = Executors.newSingleThreadExecutor();

// Future
Future<Integer> future = executor.submit(new Callable<Integer>() {
    @Override
    public Integer call() throws Exception {
        return computeValue();
    }
});

Integer result = future.get(); // Blokuje, dokud není výsledek k dispozici

// CompletableFuture
import java.util.concurrent.CompletableFuture;

CompletableFuture<Integer> completableFuture = CompletableFuture.supplyAsync(() -> computeValue());

completableFuture.thenAccept(result -> System.out.println("Result: " + result));
```

6. Executors a ThreadPools

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

// Fixní počet vláken
ExecutorService fixedPool = Executors.newFixedThreadPool(4);

// Pružný počet vláken
ExecutorService cachedPool = Executors.newCachedThreadPool();

// Jediné vlákno
ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();

// Plánované úlohy
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);
scheduler.schedule(task, 10, TimeUnit.SECONDS); // Spustí úlohu po 10 sekundách
scheduler.scheduleAtFixedRate(task, 0, 10, TimeUnit.SECONDS); // Spustí úlohu každých 10 sekund
```

Vhodné užití a příklady

Producent-Konzument

Klasický příklad vícevláknové aplikace, kde jedno vlákno (producent) vytváří data a druhé vlákno (konzument) je zpracovává.

```
import java.util.LinkedList;
import java.util.Queue;

public class ProducerConsumer {
```

```
private final Queue<Integer> buffer = new LinkedList<>();
private final int MAX_SIZE = 10;

public void produce() throws InterruptedException {
    int value = 0;
    while (true) {
        synchronized (buffer) {
            while (buffer.size() == MAX_SIZE) {
                buffer.wait(); // Čeká, když je buffer plný
            }

            buffer.add(value);
            System.out.println("Produced: " + value);
            value++;

            buffer.notify(); // Signalizuje konzumentovi
            Thread.sleep(1000); // Simulace práce
        }
    }
}

public void consume() throws InterruptedException {
    while (true) {
        synchronized (buffer) {
            while (buffer.isEmpty()) {
                buffer.wait(); // Čeká, když je buffer prázdný
            }

            int value = buffer.poll();
            System.out.println("Consumed: " + value);

            buffer.notify(); // Signalizuje producentovi
            Thread.sleep(1000); // Simulace práce
        }
    }
}

public static void main(String[] args) {
    ProducerConsumer pc = new ProducerConsumer();

    Thread producerThread = new Thread(() -> {
        try {
            pc.produce();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    });

    Thread consumerThread = new Thread(() -> {
        try {
            pc.consume();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    });

    producerThread.start();
    consumerThread.start();
```

```
    }
}
```

Moderní implementace s BlockingQueue

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class ModernProducerConsumer {
    private final BlockingQueue<Integer> buffer = new LinkedBlockingQueue<>(10);

    public void produce() throws InterruptedException {
        int value = 0;
        while (true) {
            buffer.put(value); // Blokuje, pokud je fronta plná
            System.out.println("Produced: " + value);
            value++;
            Thread.sleep(1000); // Simulace práce
        }
    }

    public void consume() throws InterruptedException {
        while (true) {
            int value = buffer.take(); // Blokuje, pokud je fronta prázdná
            System.out.println("Consumed: " + value);
            Thread.sleep(1000); // Simulace práce
        }
    }

    public static void main(String[] args) {
        ModernProducerConsumer pc = new ModernProducerConsumer();

        Thread producerThread = new Thread(() -> {
            try {
                pc.produce();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        Thread consumerThread = new Thread(() -> {
            try {
                pc.consume();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        producerThread.start();
        consumerThread.start();
    }
}
```

Paralelní výpočty

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
```

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class ParallelComputation {
    public static void main(String[] args) throws Exception {
        int numberOfThreads = Runtime.getRuntime().availableProcessors();
        ExecutorService executor = Executors.newFixedThreadPool(numberOfThreads);

        List<Future<Integer>> futures = new ArrayList<>();

        // Rozdělení úlohy na menší části
        for (int i = 0; i < 10; i++) {
            final int taskId = i;
            futures.add(executor.submit(new Callable<Integer>() {
                @Override
                public Integer call() throws Exception {
                    // Výpočet dílčího výsledku
                    System.out.println("Task " + taskId + " running on thread " +
                                       Thread.currentThread().getName());
                    Thread.sleep(1000); // Simulace práce
                    return taskId * 10;
                }
            }));
        }

        // Sběr výsledků
        int sum = 0;
        for (Future<Integer> future : futures) {
            sum += future.get(); // Blokuje, dokud není výsledek k dispozici
        }

        System.out.println("Total result: " + sum);

        executor.shutdown();
    }
}
```

Asynchronní zpracování s CompletableFuture

```java
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class AsyncProcessing {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(4);

        CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            return "Result from task 1";
        }, executor);

        CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() -> {
```

```
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            return "Result from task 2";
        }, executor);

        // Kombinace výsledků, když jsou oba k dispozici
        CompletableFuture<String> combinedFuture = future1.thenCombine(future2, (result1, result2) -
            return result1 + " + " + result2;
        });

        // Zpracování výsledku bez blokování
        combinedFuture.thenAccept(result -> {
            System.out.println("Combined result: " + result);
        });

        // Další kód může běžet souběžně...
        System.out.println("Main thread continues execution...");

        // Čekání na dokončení asynchronních úloh před ukončením programu
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        executor.shutdown();
    }
}
```

Webový server s Thread Pool

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolServer {
    private static final int PORT = 8080;

    public static void main(String[] args) throws IOException {
        ExecutorService threadPool = Executors.newFixedThreadPool(20);
        ServerSocket serverSocket = new ServerSocket(PORT);

        System.out.println("Server started on port " + PORT);

        try {
            while (true) {
                final Socket clientSocket = serverSocket.accept();
                threadPool.submit(() -> handleRequest(clientSocket));
            }
        } finally {
            serverSocket.close();
            threadPool.shutdown();
        }
```

```
    }

    private static void handleRequest(Socket clientSocket) {
        try {
            System.out.println("Handling request from " + clientSocket.getInetAddress() +
                               " on thread " + Thread.currentThread().getName());

            // Zpracování požadavku...
            Thread.sleep(1000); // Simulace práce

            clientSocket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Vhodné použití vícevláknového programování

1. **UI aplikace** - oddělení UI vlákna od pracovních vláken, aby uživatelské rozhraní zůstalo responzivní
2. **Zpracování velkého množství dat** - paralelní zpracování částí dat
3. **Síťové aplikace** - souběžné zpracování požadavků od více klientů
4. **Chatboty a komunikační aplikace** - oddělené vlákno pro odesílání a příjem zpráv
5. **Serverové aplikace** - zpracování požadavků v samostatných vláknech
6. **Dávkové zpracování** - rozdělení dávky na menší části a paralelní zpracování
7. **Real-time systémy** - zpracování dat a událostí v reálném čase
8. **Hry a multimediální aplikace** - oddělené vlákno pro renderování grafiky, fyziku, zvuk
9. **Monitorovací systémy** - paralelní sledování různých zdrojů a parametrů

Kdy nepoužívat vícevláknové programování

1. **Jednoduché úlohy** - režie spojená s vlákny může být kontraproduktivní
2. **Malé množství dat** - paralelní zpracování se nevyplatí
3. **Sekvenční úlohy** - úlohy, které vyžadují striktně sekvenční zpracování
4. **Kritické systémy s vysokými nároky na determinismus** - obtížné předvídání chování