EDICE
myslíme v ...

knihovna programátora

- Podprocesy neboli vlákna v Javě od základů k pokročilým technikám
- Použití podprocesů v programech s grafickým uživatelským rozhraním (knihovny Swing a JavaFX)
- Synchronizace a komunikace podprocesů
- Nástroje pro asynchronní výpočty

MIROSLAV VIRIUS

Java

programování podprocesů (vláken)

soubory ke stažení na
WWW.GRADA.CZ

GRADA

GRADA

EDICE
myslíme v ...

knihovna programátora

MIROSLAV VIRIUS

Java

programování podprocesů (vláken)

GRADA
Publishing

Miroslav Virius

Java – programování podprocesů (vláken)

Vydala Grada Publishing, a.s.
U Průhonu 22, Praha 7
obchod@grada.cz, www.grada.cz
tel.: +420 234 264 401
jako svou 8055. publikaci

Odpovědný redaktor Petr Somogyi
Sazba Petr Somogyi
Počet stran 200
První vydání, Praha 2021
Vytiskla TISK CENTRUM s.r.o., Moravany u Brna

ISBN 978-80-271-4254-5 (ePub)
ISBN 978-80-271-4253-8 (pdf)
ISBN 978-80-271-3266-9 (print)

Obsah

Předmluva

Kniha, kterou držíte v rukou, vás chce seznámit s programováním paralelních toků výpočtu – obvykle označovaných jako *podprocesy* neboli *vlákna* (anglicky *thread*) – v rámci jednoho programu v programovacím jazyce Java. Je to svérázná oblast programování: některé z věcí, na které jste si při programování zvykli, neplatí – nemůžete se např. vždy spolehnout na pořadí, v jakém se některé věci v programu stanou, pokud neuděláte zvláštní opatření. Může se stát, že si podprocesy – ony paralelní toky výpočtu – navzájem pokazí data, a když se jim v tom pokusíte zabránit a nedáte si pozor, zablokují se navzájem, a mohou se stát i jiné nepříjemné věci. Všechny tyto problémy mají samozřejmě řešení, o nichž budeme také hovořit.

Co od čtenáře očekávám

Očekávám, že umíte programovat v jazyce Java alespoň na základní úrovni, což znamená, že umíte napsat běžnou konzolovou aplikaci a aplikaci s grafickým uživatelským rozhraním v knihovně Swing. Znáte-li základy práce s platformou JavaFX, tím lépe. (K podrobnějšímu seznámení s Javou obecně lze použít např. česky psanou knihu R. Pecinovského [3] nebo anglicky psanou knihu [1].)

Co v této knize najdete

Tato kniha se skládá z devíti kapitol. V první kapitole se seznámíme s pojmem podproces (vlákno) a řekneme si, jak je v Javě reprezentován a jaké má vlastnosti, jak vypadá jeho životní cyklus – tedy jakými stavy může procházet. Ve druhé kapitole se seznámíme s třídou `Thread` a s rozhraním `Runnable`, naučíme se podproces vytvořit, spustit, počkat na jeho dokončení, na nějakou dobu ho uspat a podobné základní operace. Naučíme se také zacházet s přerušením podprocesů a seznámíme se s obsluhou výjimek, které se z podprocesů rozšíří.

Třetí kapitola je věnována použití podprocesů po výpočty na pozadí v programech s grafickým uživatelským rozhraním. Hovoříme jak o knihovně Swing, tak o platformě JavaFX. Abychom lépe porozuměli, jak tyto nástroje fungují, vyvineme zde také zjednodušenou analogii třídy `SwingWorker`, ovšem určenou pro platformu JavaFx.

Čtvrtá kapitola hovoří o synchronizaci podprocesů, přesněji o synchronizaci přístupu podprocesů ke sdíleným datům, a o chybách, které přitom mohou nastat. Vedle použití *zámku*, který je základním synchronizačním nástrojem, se seznámíme s atomickými datovými typy, s nástroji, které umožňují atomický přístup k neatomickým proměnným, a s knihovními datovými strukturami, jež slouží k bezpečnému předávání dat mezi podprocesy.

V páté kapitole se podíváme na komunikaci mezi podprocesy – ukážeme si, jak může jeden podproces informovat jiný, že nastala jistá událost, a jak toho lze využít. Přitom se seznámíme s tzv. *podmínkovými proměnnými* (*condition variable*) a se zámky, které se přitom používají; poznáme také nástroj pro odpočítávání, kolikrát už jistá událost nastala, a ukážeme si mj. možnou implementaci tzv. korutin v Javě. V závěru této kapitoly se krátce zastavíme u tzv. časovačů (timer).

V šesté a sedmé kapitole se budeme věnovat problematice asynchronních výpočtů, tedy výpočtů v samostatných podprocesech běžících na pozadí, jež neblokují podproces, který je spustil. Přesněji, v první z těchto dvou kapitol se seznámíme s rozhraním `Future<>`, jež představuje „budoucí výsledek" asynchronního výpočtu, a s některými dalšími nástroji – mezi jiným

i s třídou `FutureTask<>` , která je vzorovou implementací rozhraní `Future<>`. V sedmé kapitole pak budou hlavním tématem *fondy podprocesů* (*thread pool*). To jsou, zjednodušeně řečeno, nástroje pro recyklaci podprocesů a pro usnadnění práce s nimi. Nejprve si ukážeme jednoduché vlastní implementace, pak poznáme třídu `Executors`, jež poskytuje předdefinované implementace fondu podprocesů a některé další nástroje, a ukážeme si jejich použití.

V osmé kapitole se podrobněji podíváme na třídy fondů podprocesů, které jsou připraveny v knihovně Javy, seznámíme se s „továrnou na podprocesy" (rozhraním `ThreadFactory`), s paralelizací *datovodů* (*stream API*), s lokálními proměnnými v podprocesech a s dalšími nástroji, na které v předchozích kapitolách nezbylo místo. V závěru této kapitoly si povíme o významu modifikátoru `volatile` a o problematice konzistence paměti na počítačích s více procesory (resp. s více jádry procesorů).

V poslední kapitole se stručně seznámíme s předem připravenými programy. Podotýkám, že jde o programy, které neobsahují podprocesy, ale které budeme později upravovat (a při tom budeme podprocesy využívat).

Příklady

Výklad doprovází řada příkladů. Jejich zdrojové texty si můžete stáhnout na webových stránkách nakladatelství Grada Publishing (*www.grada.cz*) v sekci této knihy nebo na mých webových stránkách (odkaz na ně najdete dále). V příkladech důsledně používám české identifikátory (jedinou výjimkou je předpona `get` a `set` u přístupových metod, pokud to má význam), a to včetně háčků a čárek. Vím, že profesionální programátoři se tomu vyhýbají a že v mezinárodních týmech jsou anglické identifikátory samozřejmostí. Mám ale dlouholetou zkušenost, že při výkladu určeném začátečníkům – a mohou to být i začátečníci v práci s nějakým nástrojem, kteří jinak programovat umějí – mohou české identifikátory výrazně usnadnit orientaci v ukázkách zdrojového kódu, a to je můj hlavní cíl. Má to ale ještě jednu přednost: v příkladech to umožňuje snadno rozlišit, co jsou třídy z knihovny a co naše vlastní, popřípadě které metody jsou zděděné po předcích z knihovny a které jsou naše vlastní.

Připravené programy

Mnoho příkladů v této knize je založeno na různých variantách úlohy *n* dam. To je klasická kombinatorická úloha formulovaná přibližně v polovině 19. století. Máme šachovnici o $n \times n$ polích a naším úkolem je umístit na ni *n* dam tak, aby se podle šachových pravidel neohrožovaly. Mezi soubory ke stažení je proto v adresáři `VýchozíProgramy\NDam` připravena třída `Řešitel`, jež obsahuje metody pro nalezení všech řešení této úlohy, pro postupné nacházení jednotlivých řešení, pro nalezení řešení, která mají první dámu na zadané řádce apod. Ovšem hned v prvním příkladu ve 2. kapitole si ukážeme, že tato třída, skvěle fungující v „obyčejném" programu, nemusí fungovat správně v programu s více podprocesy. Proto v dalších příkladech používáme opravenou verzi, kterou najdete v adresáři `VýchozíProgramy\UpravenýŘešitel`.

Vedle toho zde najdete dva programy, které zobrazují řešení úlohy *n* dam v okně; první, založený na knihovně Swing, je uložen v adresáři `VýchozíProgramy\NDamSwing`, a druhý, určený pro platformu JavaFX, najdete v adresáři `VýchozíProgramy\NDamFX`. Oba pochopitelně využívají třídu `Řešitel` po opravě.

Poslední z připravených příkladů je program na kreslení únikových fraktálů, jako je Mandelbrotova nebo Juliova množina. Najdete ho v adresáři `VýchozíProgramy\fraktál`.

Žádný z těchto programů nevyužívá podprocesy – přesněji, nevyužívá je na úrovni, o níž zde hovoříme. (Toto tvrzení vysvětlíme a upřesníme ve 3. kapitole.) Úpravy těchto programů, aby využívaly na různých místech podprocesy, jsou předmětem řady příkladů.

Použité nástroje

Všechny příklady jsem odladil v Javě 15 (v době psaní této knihy to byla aktuální verze). Přitom jsem používal převážně vývojové prostředí IntelliJ Idea 2020.2 Community Edition; vedle toho jsem používal prostředí Apache NetBeans 12.1, občas ale také starší verze až po NetBeans 8.2. V některých případech jsem vyzkoušel i starší verze Javy, a to až po Javu 8 včetně; s tím se setkáme především na konci 8. kapitoly. (Ovšem pozor, zdrojové kódy je při použití starších verzí Javy třeba upravit – nejde jen o klíčové slovo `var`, které přišlo s verzí 9, ale i o použití některých metod a tříd.)

Terminologie

V celé knize používám důsledně českou terminologii. Vím, že má mnoho odpůrců, jsem však přesvědčen, že použití vhodných českých názvů výrazně usnadní pochopení, oč jde. Anglické termíny samozřejmě uvádím alespoň při prvním výskytu také.

Poznámka k literatuře

Specifikace tříd a metod z knihoven Javy, které v této knize uvádím, jsem čerpal z dokumentace k JDK 15 [4] a k JavěFX 15 [5]. Tyto odkazy již v textu zpravidla neopakuji.

Základní verze programu na výpočet fraktálů, který používám v řadě příkladů, vznikla úpravou jedné z počátečních verzí aplikace, jejíž vývoj popisuji ve skriptu [9] určeném posluchačům Fakulty jaderné a fyzikálně inženýrské ČVUT v Praze.

Na závěr

I přes veškerou péči, kterou jsem této knize věnoval, se do ní mohly vloudit chyby. Jestliže v této knize nějakou najdete, dejte mi prosím zprávu na níže uvedenou adresu; bude-li to možné, uveřejním na svých webových stránkách opravu.

V Praze, 21. února 2021

Miroslav Virius
miroslav.virius@fjfi.cvut.cz
http://people.fjfi.cvut.cz/virius

1 Podproces neboli vlákno

V této kapitole se seznámíme s pojmem *podproces* (*vlákno*), s jeho životním cyklem a se stavy, kterými v průběhu života prochází. Zatím ještě nebudeme programovat, s tím začneme až v následující kapitole.

1.1 Proces a podproces

Jistě víte, že v počítačové terminologii se pod pojmem *proces* rozumí instance programu, kterou provádí počítač a kterou spravuje operační systém. Každý proces má svou operační paměť, ve které jsou uloženy jak prováděné instrukce tvořící program, tak data, s nimiž program pracuje. Proces má také přiděleny další prostředky spravované operačním systémem – vstupní a výstupní zařízení apod.

Na dnešních počítačích může běžet zároveň více procesů, tj. může být zároveň spuštěno a prováděno více programů. Přitom jednotlivé procesy jsou operačním systémem navzájem izolovány; v prvním přiblížení můžeme tvrdit, že žádný z běžících procesů neví o ostatních,[1] takže mezi nimi nedochází ke konfliktům – nepřepisují si data, nenastávají konflikty, pokud jde o vstupy a výstupy apod.

Proces, tvořený programem napsaným v Javě, se může rozdělit na několik paralelně běžících *podprocesů* (anglicky *thread*).

Anglický termín *thread* bývá dnes zpravidla překládán jako *vlákno*; lze se setkat i s termíny *tok výpočtu* nebo *podproces*. I když je mi jasné, že se tím odchýlím od terminologie používané ve většině současných českých publikací, budu používat poslední možnost, označení *podproces*, neboť velice přesně popisuje, o co jde.

1.1.1 Podproces

Podproces je tedy jeden ze souběžných toků výpočtu v rámci jednoho procesu. Má-li proces k dispozici dostatečný počet procesorů, může každý podproces běžet na jednom z nich; jestliže ne, jestliže je více podprocesů než logických procesorů,[2] musí se podprocesy na jednotlivých procesorech střídat. Každý podproces pak běží po vymezenou – zpravidla velmi krátkou – dobu, poté je odstaven a procesor je přidělen jinému podprocesu. Tak i na počítači s jedním procesorem s jediným jádrem vznikne dojem, že podprocesy běží opravdu paralelně.

O přidělování a odebírání procesoru se stará tzv. plánovač (scheduler). Typicky se jedná o součást operačního systému. Ovšem plánovač je i součástí virtuálního stroje Javy (JVM), takže

1 Zopakujme si, že jde o první přiblížení. Ve skutečnosti existují možnosti, jak poslat data jinému procesu, jak zjistit, zda běží jiná instance téhož programu apod. Pro náš výklad to ale není podstatné.

2 U současných počítačů funguje zpravidla každé jádro jako dvojice logických procesorů.

program v Javě s více podprocesy může běžet i pod operačním systémem, který podprocesy nepodporuje; to se týká nejen historického operačního systému DOS, ale i starších verzí Linuxu a operačních systémů některých starších mobilních telefonů.

Jestliže operační systém, na kterém náš program běží, podporuje podprocesy, spolupracuje plánovač JVM s plánovačem operačního systému.

Kooperativní a preemptivní plánování

Pro přidělování procesoru podprocesům lze použít dvě základní strategie [3]: *preemptivní* a *kooperativní plánování*.

Při kooperativním plánování přidělí plánovač jednomu podprocesu procesor; podproces nějakou dobu běží a pak se procesoru dobrovolně vzdá, aby ho mohl plánovač přidělit jinému podprocesu. Kooperativní plánování tedy vyžaduje aktivní spolupráci podprocesů (nevzdá-li se podproces procesoru, zablokuje ostatní podprocesy), na druhé straně klade menší nároky na hardware počítače.

Při preemptivním plánování přidělí plánovač vybranému podprocesu procesor a po nějaké době mu ho zase odebere a přidělí ho dalšímu. Preemptivní plánování je bezpečnější než kooperativní plánování a nevyžaduje spolupráci podprocesů, vyžaduje ale procesor, který k tomu poskytuje potřebné nástroje.

Podprocesy a plánování v Javě

V raných verzích Javy, jež poskytovaly podprocesy i pod operačními systémy, které samy podprocesy nepodporovaly, byly podprocesy implementovány na úrovni JVM, takže nezávisely na operačním systému a byly implementovány na základě kooperativního plánování (to slibovala specifikace jazyka). V současných verzích Javy představují podprocesy pouze obal podprocesů operačního systému a při jejich implementaci se využívá preemptivní plánování. V některé z příštích verzí by se měly do Javy vrátit i podprocesy na úrovni JVM, nezávislé na podpoře operačního systému. Odborné články o nich v současné době hovoří jako o *virtuálních podprocesech* nebo o *vláknech* (*fiber*) [12]. Podrobněji o nich v této knize nebudeme hovořit, neboť jsou teprve ve studiu příprav.

1.1.2 Podproces není izolován

Jak víme, procesy jsou operačním systémem navzájem izolovány, takže mezi nimi v běžných situacích nedochází ke konfliktům, pokud jde o data nebo prostředky operačního systému. Na druhé straně podprocesy v Javě, které jsou součástí jednoho procesu, mohou všechny přistupovat k týmž datům, k týmž vstupním a výstupním zařízením atd. To znamená, že zde může docházet ke konfliktům: Nedáme-li si pozor, budou si podprocesy přepisovat data, budou vznikat zmatené výstupy atd.

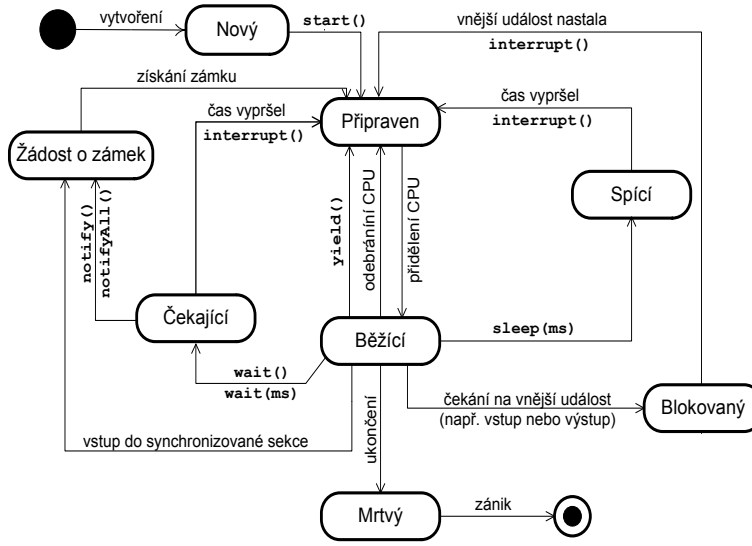
O tom, jak se takovýmto a dalším problémům vyhnout, budeme hovořit především ve 4. kapitole.

Poznamenejme, že každý program v Javě má alespoň jeden podproces, a to ten, v němž běží metoda `main()`. Může jich však mít více, i když sami explicitně žádný další nevytvoříme. To se týká např. programů s grafickým uživatelským rozhraním, v nichž – zjednodušeně řečeno – okna běží v samostatném podprocesu.

O podprocesu, v němž běží metoda `main()` programu, budeme hovořit jako o *hlavním podprocesu*.

1.2 Životní cyklus podprocesu

Podproces v Javě je reprezentován instancí třídy `java.lang.Thread`. K jeho vytvoření typicky použijeme konstruktor (o tom budeme hovořit v následující kapitole). Po vytvoření plní podproces nějakou dobu své úkoly a nakonec zanikne; mezitím ovšem projde řadou stavů. Diagram, který ukazuje tyto stavy a možné přechody mezi nimi, vidíte na obrázku 1.1.



Obrázek 1.1 Diagram stavů podprocesu v Javě

Obrázek 1.1 představuje tzv. *stavový diagram* (*state diagram*, *state machine diagram*) v modelovacím jazyce UML. Tento diagram vždy začíná vyplněným kruhem a končí „terčíkem". Jednotlivé stavy popisované entity – v našem případě podprocesu – jsou vyjádřeny ovály. Šipky ukazují možné přechody mezi těmito stavy; k nim jsou připsány události, které takovýto přechod způsobí – mimo jiné i volání určitých metod třídy `Thread`. S metodami zmíněnými v tomto obrázku se seznámíme v následujících kapitolách.

Tento diagram jsem s drobnými úpravami převzal z dnes již neexistujících webových stránek Ing. Rudolfa Pecinovského, CSc. Na rozdíl od podobných diagramů, které lze najít v odborné literatuře, rozlišuje stavy *Připraven* a *Běžící*, což je pro pochopení skutečného fungování aplikace s paralelně běžícími podprocesy v některých situacích velice důležité.

V následujícím výkladu se setkáte s pojmy, jako je *zámek*, o nichž budeme hovořit podrobně později. Jestliže některým podrobnostem při prvním čtení neporozumíte, nevadí, vše bude podrobně vysvětleno v následujících kapitolách.

1.2.1 Stavy podprocesu a přechody mezi nimi

Podívejme se na jednotlivé stavy podprocesu a na přechody mezi nimi. Zopakujme si, že na rozdíl od většiny publikací rozlišujeme stavy *Připraven* a *Běžící*.

Zde si povíme základní informace, které jsou nezbytné pro chápání výkladu; v následujících kapitolách se samozřejmě podíváme na jednotlivé možnosti podrobněji.

Nový podproces

Po vytvoření konstruktorem je podproces ve stavu *Nový*. V tomto stavu nic nedělá, pouze čeká na spuštění voláním metody `start()`. Po zavolání této metody přejde do stavu *Připraven*.

Připravený a běžící podproces

Podproces ve stavu *Připraven* čeká na přidělení procesoru. Neběží, tedy nevykonává žádné instrukce. Do stavu *Běžící*, kdy vykonává instrukce programu, může přejít jedině tehdy, když mu plánovač přidělí procesor (někdy se také říká, že mu přidělí *čas procesoru*). Když se plánovač rozhodne podprocesu odebrat procesor, aniž by tento podproces dokončil výpočet (a aniž by nastala některá z událostí, o nichž budeme hovořit dále), převede ho plánovač zpět do stavu *Připraven*. Také jestliže v běžícím podprocesu zavoláme metodu `yield()`, bude podproces převeden do stavu *Připraven*. (Voláním této metody se podproces dobrovolně vzdá přiděleného procesoru.)

Mrtvý podproces

Jestliže v době, kdy má podproces přidělen procesor, dokončí vše, co měl udělat, přejde do stavu *Mrtvý*. Mrtvý podproces nelze oživit, odpovídající instance třídy `Thread` už jen čeká na odstranění automatickou správou paměti (garbage collector).

Blokovaný podproces

Jestliže se podproces dostane do situace, kdy bude čekat na nějakou vnější událost, např. na vstup od uživatele, na navázání síťového připojení apod., přejde do stavu *Blokován*. Blokovanému podprocesu není přidělován procesor. Z tohoto stavu může přejít do stavu *Připraven* buď poté, co očekávaná událost nastane, nebo tím, že jiný podproces zavolá jeho metodu `interrupt()`, a tak ho *přeruší*.

O důsledcích přerušení budeme hovořit dále.

Spící podproces

Jestliže v běžícím podprocesu zavoláme metodu `sleep()`, přejde do stavu *Spící*. Při volání této metody zadáváme časový interval, po který spánek potrvá. Také spícímu podprocesu není pochopitelně přidělován procesor. Spánek skončí poté, co vyprší zadaný časový interval, nebo poté, co jiný podproces zavolá jeho metodu `interrupt()` a tak ho *přeruší*.

Podproces čekající na vyrozumění

Voláním metody `wait()` přejde podproces do stavu *Čekající*, v němž čeká na vyrozumění, že nastala v jiném podprocesu jistá událost. O této události může být vyrozuměn voláním metody `notify()` nebo `notifyAll()`; pak přejde do stavu *Žádost o zámek*, v němž bude čekat na přidělení zámku, tedy na povolení přístupu ke sdíleným datům nebo jiným prostředkům. (Podrobněji o tom budeme hovořit v 5. kapitole.)

Jestliže jiný podproces zavolá jeho metodu `interrupt()`, bude čekající podproces přerušen a přejde do stavu *Připraven*.

Při volání metody `wait()` lze také zadat časový interval, po který má volající podproces čekat; jestliže tento interval vyprší, aniž je čekající podproces vyrozuměn voláním metody `notify()` nebo `notifyAll()`, přejde do stavu *Žádost o zámek*.

Žádost o zámek pro přístup ke sdíleným prostředkům

Jak víme, podprocesy mohou sdílet data a prostředky operačního systému. To znamená, že mohou např. pracovat s týmiž proměnnými; přitom se může stát, že se jeden podproces pokusí číst data, která právě jiný podproces mění. Přečtená data pak nebudou konzistentní, naopak – budou nejspíš zcela nesmyslná. Proto se sdílená data uzavírají do tzv. synchronizovaných sekcí, úseků kódu, do nichž může podproces vstoupit, jen když k němu získá zámek (lock). (Přesnější by bylo říkat, že získá klíč, kterým si otevře zámek položený na vstup do tohoto úseku kódu, ale říká se, že získá zámek.)[3] To znamená, že když chce podproces do takovéhoto úseku vstoupit, musí požádat o zámek – přejde do stavu *Žádost o zámek* a v tomto stavu čeká (není mu přidělován procesor), dokud se zámek neuvolní.

V okamžiku, kdy se zámek uvolní, takže ho žádný jiný podproces nevlastní, může ho žádající podproces získat a přejít do stavu *Připraven*.

Po dokončení operací se sdílenými prostředky musí podproces zámek (vlastně klíč k zámku, ale říká se zámek) uvolnit, vrátit ho, aby ho mohl získat další žadatel.

Jestliže zavoláme metodu `interrupt()` podprocesu čekajícího na zámek, bude přerušen a také přejde do stavu *Připraven*. Se základním druhem zámku v Javě se seznámíme ve čtvrté kapitole, v páté kapitole poznáme ještě další možnosti.

1.2.2 Přerušení podprocesu

Je-li podproces ve stavu *Blokovaný*, *Čekající*, *Spící* nebo *Žádost o zámek*, přejde – zavolá-li jiný podproces jeho metodu `interrupt()` – do stavu *Připraven*. Ovšem přitom vznikne výjimka typu `java.lang.InterruptedException`.

Jestliže tuto metodu zavoláme pro podproces, který je ve stavu *Běžící* nebo *Připraven*, uvedená výjimka nevznikne. Bude však nastaven příznak přerušení, který lze v těle běžícího podprocesu zjistit voláním metody `interrupted()` nebo `isInterrupted()`; k tomu se vrátíme později.

1.2.3 Běh a připravenost podprocesu

Už jsme si řekli, že po spuštění, po probuzení ze spánku, po skončení blokování a po získání zámku přejde podproces do stavu *Připraven*, a teprve když mu plánovač přidělí procesor, opravdu se rozběhne. Přitom čas, který uběhne mezi přechodem do stavu *Připraven* a skutečným rozběhnutím, závisí na počtu procesů a jejich podprocesů, které aktuálně čekají na přidělení procesoru, a na dalších okolnostech, takže se může – a bude – při různých běžích téhož programu lišit. To má některé zajímavé důsledky. Například:

- Jestliže spouštíme z jednoho podprocesu několik dalších, bude pořadí, ve kterém se spouštěné podprocesy rozeběhnou, a pořadí, v jakém se pak budou v běhu střídat, v podstatě nepředvídatelné.
- Jestliže uspíme podproces na zadanou dobu *t*, máme jistotu, že bude nečinný *nejméně* po dobu *t* (nebude-li mezi tím např. přerušen), nemáme ale jistotu, že se znovu rozběhne ihned po uplynutí doby *t*.

3 Možná by bylo vhodnější hovořit o *žádosti o přístup* a vyhnout se úvahám o zámcích a klíčích; zůstaneme ale u vžité terminologie a budeme hovořit o *žádosti o zámek*.

1.2.4 Identifikace podprocesu

Každý podproces běžící pod JVM dostane při vytvoření číslo, které ho identifikuje; v dokumentaci je označováno *thread ID*. Jedná se o kladnou hodnotu typu `long`, která je po dobu života podprocesu jednoznačná; po zániku podprocesu může toto číslo JVM znovu použít pro jiný podproces. Z programu lze toto identifikační číslo zjistit voláním metody `getID()`.

Jako parametr konstruktoru třídy `Thread` můžeme také zadat znakový řetězec a přidělit tím podprocesu jméno. (Jestliže ho nezadáme, dostane při vytvoření nějaké implicitní jméno.) Toto jméno lze později zjistit voláním metody `getName()`.

1.2.5 Zavržené stavy a možnosti

Podproces lze *pozastavit* – převést do stavu *Pozastaven* – voláním metody `suspend()`. Pozastavený podproces neběží a čeká na obnovení voláním metody `resume()`, která ho převede zpět do stavu *Připraven*. Obě tyto metody jsou v dnešní Javě pokládány za zavržené (deprecated), neboť nezacházejí korektně se zámky, které podproces v okamžiku volání metody `suspend()` vlastnil. V některé z příštích verzí mají být z Javy odstraněny. Z tohoto důvodu není stav *Pozastaven* ani znázorněn na diagramu na obrázku 1.1.

Další zavrženou možností je okamžité ukončení podprocesu voláním metody `stop()`; důvodem je opět nekorektní zacházení se zámky, které podproces v okamžiku volání této metody vlastnil. Proto není ani tato možnost na diagramu na obrázku 1.1 znázorněna.

1.2.6 Ukončení procesu

Statická metoda

```
public static void exit(int kódUkončení);
```

třídy `java.lang.System` ukončí virtuální stroj Javy, v němž běží program, z něhož byla tato metoda volána. Tím automaticky zaniknou i podprocesy, které v rámci tohoto programu běžely.

1.3 Bezpečnost v prostředí s více podprocesy

V následujících kapitolách se budeme setkávat s tvrzením, že daná třída je bezpečná v prostředí s více podprocesy, že je zabezpečena pro použití v prostředí s více podprocesy apod. – anebo naopak, že v takovém prostředí bezpečná není. I když je vám nejspíš intuitivně jasné, oč jde, neuškodí, když toto tvrzení alespoň trochu upřesníme.

Jak uvidíme dále, jestliže několik podprocesů přistupuje k týmž datům, může dojít ke konfliktu, protože jeden podproces se může pokusit číst data, která právě jiný podproces mění. Výsledkem bude, že podproces, který data čte, získá nesmyslné hodnoty. Java nabízí různé způsoby, jak se takovýmto konfliktům vyhnout; obvykle se hovoří o synchronizaci přístupu ke sdíleným datům.

Některé třídy jsou naprogramovány tak, že při jejich použití nemůže k takovýmto konfliktům dojít – a to je obsahem sdělení, že jsou bezpečné v prostředí s více podprocesy. Ovšem pokud v dokumentaci takovouto informaci nenajdeme, musíme předpokládat, že daná třída v prostředí s více podprocesy bezpečná není.

Podrobněji budeme o problematice synchronizace přístupu ke sdíleným datům hovořit ve 4. kapitole.

1.4 Kolik podprocesů použijeme?

Budeme-li paralelizovat výpočet, je třeba uvážit, kolik podprocesů použijeme. Přitom záleží na tom, proč vlastně výpočet paralelizujeme. Podívejme se na některé možné důvody:

- Chceme maximálně využít kapacitu procesoru počítače, na kterém program poběží.
- Potřebujeme zabezpečit „responsivitu" (říká se také „reaktivitu") programu – tedy aby program i po dobu náročného výpočtu reagoval na podněty uživatele.
- Program musí souběžně reagovat na podněty z několika různých zdrojů (např. při řízení technologického procesu).
- A další…

První důvod, snaha o maximální využití prostředků, které daný počítač poskytuje, se v současné době stává jedním z nejčastějších důvodů paralelizace programů. Přitom si musíme uvědomit, že nejefektivnější náš program bude, použijeme-li tolik podprocesů, kolik bude mít virtuální stroj (JVM) k dispozici procesorů[4] pro běh našeho programu. Použijeme-li jich více, přibude jen „administrativa" spojená s přepínáním podprocesů, takže ve výsledku se tím program zpomalí.

1.4.1 O kolik náš program zrychlí, použijeme-li více procesorů?

Některé problémy lze řešit rychleji, použijeme-li více procesorů, jiné jsou však ze své podstaty sériové – *nejprve* je třeba vyřešit rovnici, *pak* můžeme její řešení využít. U některých úloh může přidání několika procesorů výpočet urychlit, ale přidání dalších už k dalšímu zrychlení nepovede. Máme-li za úkol vyřešit pět navzájem nezávislých rovnic, můžeme k tomu využít pěti procesorů. Jestliže ovšem řešení jednotlivých rovnic neumíme paralelizovat, přidání dalších procesorů už ke zrychlení nepovede.[5]

Jinak řečeno, je-li důvodem pro použití více podprocesů zrychlení výpočtu, musíme umět zadanou úlohu paralelizovat, rozložit na podprocesy, jejichž počet bude odpovídat počtu procesorů, které máme k dispozici.

Program obvykle nelze paralelizovat plně – můžeme paralelizovat pouze některé části. Jakého zrychlení pak můžeme dosáhnout? Na tuto otázku odpovídá tzv. *Amdahlův zákon* (*Amdahl's law*, v anglicky psané literatuře označovaný také jako *Amdahlův argument*, *Amdahl's argument*).

1.4.2 Amdahlův zákon

Je-li F velikost části výpočtu, kterou je třeba provést sériově, a máme-li k dispozici N procesorů, lze maximální zrychlení S, kterého lze paralelizací dosáhnout, vyjádřit vztahem:

$$Z \leq \frac{1}{F + \frac{1 - F}{N}}$$

4 Pozor, to neznamená nutně počet procesorů na daném počítači, ale počet logických procesorů, které jsou našemu programu k dispozici. Java poskytuje nástroj, který umožňuje tento počet za běhu programu zjistit; seznámíme se s ním v následující kapitole.

5 Podobný příklad „ze života": Jestliže jeden kopáč vykope požadovaný příkop o délce 10 m za jeden pracovní den, vykopou ho dva kopáči za půl dne a čtyři kopáči za dvě hodiny. To ovšem neznamená, že týž příkop vykope čtyřicet kopáčů za dvanáct minut. Umlátili by se navzájem krumpáči, a tak nejspíš čtyři pětiny z nich odejdou s tichým souhlasem mistra do hospody.

Když počet procesorů *N* poroste, bude se hodnota tohoto zlomku blížit k 1/*F*, bude však stále menší. To znamená, že lze-li paralelizovat nejvýše polovinu výpočtu (*F* = 0,5), bude maximální zrychlení, kterého lze dosáhnout s velmi velikým počtem procesorů, nejvýše dvojnásobné (*S* = 2).

Z Amdahlova zákona ale také plyne, že velký počet procesorů nemusí být ekonomický. Máme-li k dispozici 2 procesory (a stále předpokládáme *F* = 0,5, tedy že paralelizovat lze nejvýše polovinu výpočtu), můžeme dosáhnout zrychlení *Z* = 4/3 = 1,33. Jinak řečeno, můžeme zkrátit výpočet na ¾ původní doby. Při 8 procesorech se výpočet může zkrátit na 9/16 = 0,56 původní doby, při 20 procesorech to bude zkrácení na 0,5375 původní doby. Ovšem počítač s osmi virtuálními procesory je v současné době běžný, počítač s 20 nebo více procesory běžný zatím není (takže je dosti drahý), a přitom rozdíl v urychlení výpočtu mezi 8 a 20 procesory je pouze 0, 0225, tedy ve většině případů zanedbatelný.