Diplomová práce - rešerše

Abstrakce

Počítače jsou jedny z nejsložitějších systémů, které člověk kdy vytvořil. Komplexita celého počítače dalece přesahuje lidské schopnosti, lidský mozek prostě není schopen pojmout na jednou tak obrovské množství informací. Vypořádáváme se s tím pomocí abstrakce. Začíná to elektrony ve vodiči, které si předávají elektrický náboj. Pro zjednodušení se na ně díváme jako proud který se přelévá vodičem sem a tam. Pomocí tranzistoru můžeme sestavit elektronické obvody v kterých proud plynule teče pomocí nám známých zákonů. Vyjádření logické funkce je však pomocí takového obvodu nesnadné. Pokud, ale představíme další úroveň abstrakce a z tranzistorů vytvoříme logická hradla, kde na každém drátu nám určité napětí představuje hodnotu jedna nebo nula stane se vytvoření řešiče nějaké logické formule snadným. Procesor, nejdůležitější část počítače, je velmi složitý elektronický obvod. Na venek se však tváří jako černá skřínka a jediné co potřebujeme pro komunikaci s ním znát je jeho ISA (instrukční sada). ISA je na rozhraní mezi softwarem a hardwarem. Nad ni je umístěn Operační systém, který poskytuje uživateli příjemné rozhraní pro ovládání počítače a také knihovny pro usnadnění práce aplikačních programů.

Jak je vidět celý počítač je protknout mnoha úrovněmi abstrakce. A to nejen těmi výše zmíněnými, ale i mnoha dalšími, tento princip je využit skoro všude. Jeho obrovskou výhodou je, že lidem umožňuje přemýšlet čistě na úrovni abstrakce, která je zajímá a ostatní nebrat v potaz. Pokud píšeme program nemusí nás zajímat jaké napětí bude na vodičích procesoru, na jakém procesoru běží a často dokonce ani na jakém operačním systému. Druhou a také velmi důležitou výhodou tohoto principu je, že nejsme závislý na tom co konkrétně je na nižší úrovni abstrakce. Tyto úrovně jsou odděleny rozhraním a pokud ten v nižší úrovni toto rozhraní dodržuje a poskytuje nemusí nás zajímat jak interně řeší vykonávání našich příkazů. Program který je pak napsán pro nějaké rozhraní bude fungovat na každém stroji který toto rozhraní splňuje. Dobře definované rozhraní umožňuje aby například vývojáři Microsoftu programovali operační systém nad rozhraním, konkrétně instrukční sadou IA-32 a zároveň v Intelu takový procesor vyráběli bez toho aby mezi sebou museli jakkoliv komunikovat. Podobné je to s operačním systémem a rozhraním které poskytuje aplikačním programům.

Jako nevýhodu lze vidět to že program napsaný pro jedno rozhraní nebude fungovat na druhém. Program napsaný pro Windows nebude fungovat pod Linuxem. Program běžící nad instrukční sadou IA-32 nebude běžet nad ARM. Obecně je rozmanitost v rozhraních dobrá, podporuje inovace a brání stagnaci. Omezuje však použitelnost softwaru, cože je omezující obzvláště v době internetu, kdy je výhodné používat software svobodně jako data.

Abstrakce nad pamětí a I/O (vstupy a výstupy) odstranila většinu závislostí na konkrétním hardwaru, ale některé závislosti stále zůstávají. Mnoho operačních systému je vyvinuto pro konkretní systémovou architekturu. Například pro jeden procesor nebo pro více jádrový procesor se sdílenou pamětí. Běžně se předpokládá, že hardwarové zdroje systému jsou

využívány jen jedním operačním systémem. Toto váže všechny hardwarové zdroje k jednomu systému. Omezuje to systém a omezuje flexibilitu, bezpečnost a izolaci chyb. Obzvláště pokud je systém využíván více uživateli.

Virtualizace umožňuje výše zmíněná omezení minimalizovat a zvýšit flexibilitu. Virtualizovaný systém se na venek jeví stejně jako ten reálný. Tedy splňuje domluvené rozhraní, ale uvnitř všechny zdroje a příkazy z rozhraní mapuje na opravdový systém pod ním. V tomhle aspektu je virtualizace podobná abstrakci, obě implementují rozhraní nad sebou a využívají nějaké jiné rozhraní. Rozdíl je v tom že virtualizace nemusí nutně skrývat detaily. Úroveň detailu virtualního rozhraní a toho reálného které se nakonec použije mohou být stejné.

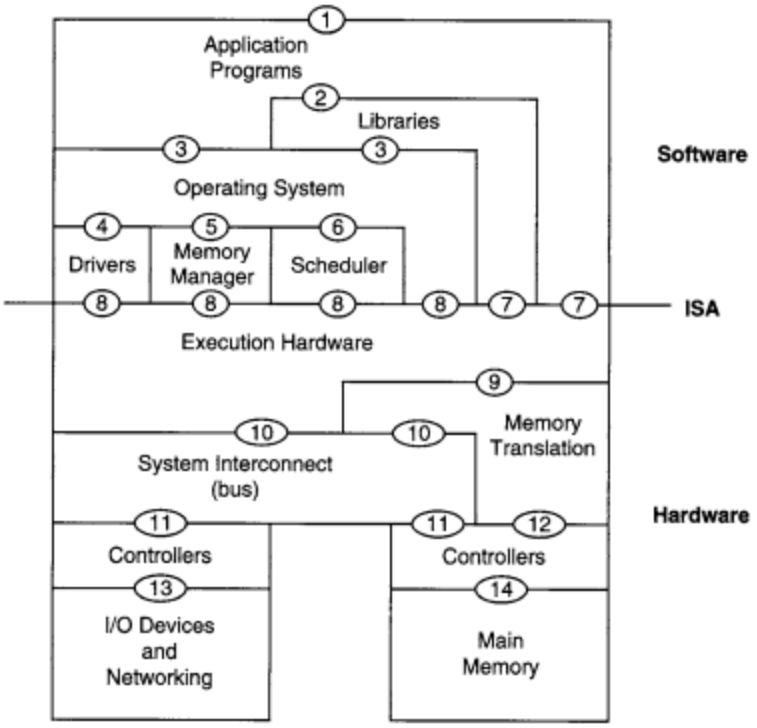
Jako příklad virtualizace si lze představit pevný disk. Pokud si uživatel chce disk rozdělit na více částí, je to možné a tyto části se opravdu budou na venek tvářit jako disky dva. Rozhraní těchto disků však bude identické s diskem původním. Nedošlo k žádnému snížení detailů. Jedná se tedy jen o virtualizaci nikoliv o abstrakci.

Virtualizace se nemusí omezovat jen na jednotlivou komponentu jako je pevný disk, ale lze virtualizovat i celý stroj. Takový software se označuje pojmem virtuální stroj (VM - virtual machine). Takový virtuální stroj pak umožňuje třeba spustit v něm program pro Windows i když reálný operační systém je Mac.

Existuje mnoho druhů virtuálních strojů s mnoha účely. Virtuální stroj může běžet hned nad ISA a umožňovat běh dvou operačních systému najednou. Může emulovat jednu ISA a ve skutečnosti používat úplně odlišnou. Virtuální stroj může být i aplikační program, který vám umožní mít puštěný Linux přímo ve Windows. Vyšší programovací jazyky také často běží na nějakém virtuálním stroji. Existuje virtuální stroj jenž funguje pouze jako transparentní vrstva a provádí optimalizace kódu něž se spustí na stroji opravdovém.

Architektura počítače

S virtuálními stroji souvisí architektura počítače. Jedna z věcí která nás u nich nejvíce zajímá je jak dobře implementují rozhraní dané právě architekturou. Architektura budovy určuje jak se budova bude jevit jejím uživatelům a jaké možnosti jim poskytne. Nepopisuje už z jakých cihel bude postavena ani detaily elektrických rozvodů. Podobně i architektura počítačového systému popisuje jak se jeví jeho uživatelům a jaké funkce jim poskytuje. Ne už jeho konkrétní implementaci. Každá architektura může mít mnoho implementací s rozdílnými charakteristikami. Například výkonnou implementaci a implementaci s nízkou spotřebou.



Na přiloženém obrázku je vidět typické rozdělení vrstev abstrakce. Každá z těchto vrstev má svojí architekturu a implementaci. Virtuální stroje se typicky vyskytují kolem hranice mezi hardwarem a softwarem to jest ISA nebo výše. Dvě části ISA jsou důležité při specifikaci virtuálního stroje. První je část viditelná aplikačním programům (rozhraní č.7 na obrázku) neboli uživatelská ISA. Druhá část je ISA kterou používá operační systém ke správě zdrojů neboli systémová ISA. Operační systém, ale může přistupovat k oběma (rozhraní č.8 na obrázku).

Další pro virtuální stroje důležitá rozhraní jsou ABI (application binary interface) které v sobě zahrnuje rozhrání 3 a 7 na obrázku a API (application programming interface), které se skládá z rozhraní 2 a 7.

ABI je rozhraní s kterým program komunikuje za běhu, tedy po tom co je přeložen do strojového kódu. Definuje například jak se předávají parametry funkcím nebo kde se bude nacházet návratová hodnota. Programy kompilované pro konkrétní ABI mohou běžet pouze počítači se stejnou ISA a operačním systémem.

API je rozhraní se kterým se dostane do kontaktu i aplikační programátor. Patří sem standardní knihovny které může program využít pro volání služeb operačního systému. Aplikaci napsanou pro určité API je možné zkompilovat pro jakýkoliv systém který dané API implementuje.

Virtuální stroje

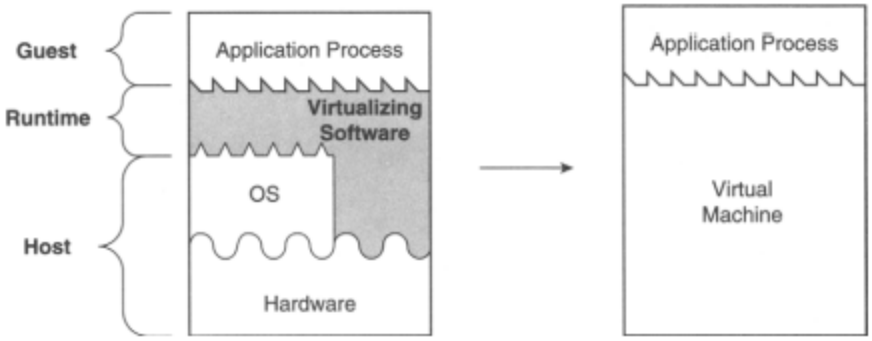
Stroj může v tomto kontextu znamenat více věcí. Pro proces vykonávající nějaký program je to jemu přidělená paměť spolu s registry a instrukcemi které může provádět. Proces nemůže k vstupům a výstupům přistupovat jinak než pomocí systému. Většinou proto používá nějaké knihovny, jejichž kód se spustí v rámci jeho vykonávání. Rozhraní mezi procesem a strojem představuje ABI.

Z pohledu operačního systému je celý systém spuštěn na stroji. Paměť s kterou OS pracuje je mu přidělena trvaleji a zatímco procesy začínají a končí operační systém k ní má stále přístup. Rozhraním mezi operačním systémem a strojem je ISA.

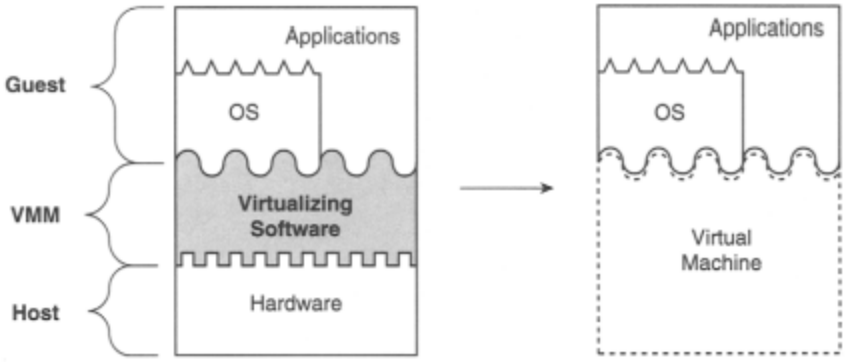
Virtuální stroj je pak nejen software vykonávající virtualizaci ale i skutečný hostitelský stroj na kterém virtuální stroj běží. Samotný spuštěný proces/systém to samozřejmě o hostitelském systému nic neví. Podle toho jak rozdílně se jeví stroje procesu a operačnímu systému, lze rozdělit i virtuální stroje na procesní virtuální stroje a systémové virtuální stroje.

Procesní virtuální stroj je často nazýván runtime, cože je zkráceně runtime software. Takový VM je spuštěn spolu s daným procesem, poskytuje mu prostředky pro komunikaci s I/O zařízeními a hardwarem celkově a když proces skončí je ukončen i VM.

Naproti tomu systémový virtuální stroj poskytuje celé prostředí na které může být nainstalován operační systém a běžet velmi dlouho. Na obrázku je vidět systémový virtuální stroj, který emuluje jednu ISA na jiné. Jeho virtualizačnímu softwaru se také někdy říká VMM (Virtual Machine Monitor).



Procesní virtuální stroj



Systémový virtuální stroj

Virtuální stroje pro jazyky vyšší úrovně

Virtuální stroje pro jazyky vyšší úrovně zkráceně HLL VM jsou procesní virtuální stroje, které slouží jako mezivrstva mezi operačním systémem a daným jazykem. Umožňují programům aby nebyli závislé na konkrétním operačním systému a konkrétní ISA. Bez virtuálního stroje by se program musel v lepším případě pro každý OS a ISA znovu zkompilovat a někdy by se museli přepisovat i knihovny nebo volání OS. Tyto Stroje zpravidla nemají emulovat nějakou konkrétní architekturu, ale jsou navrženy tak aby co nejlépe odpovídali potřebám vyšších (často objektových) jazyků. Nicméně pro některé existují i jejich hardwarové implementace. Knihovny těchto strojů poskytují vyšší stupeň abstrakce, než klasické systémové knihovny.

Ve zbytku tohoto textu, pokud zmíním virtuální stroj bez dalšího přívlastku budu mít na mysli vždy procesní virtuální stroj tohoto typu.

Protože cílem VM je nezávislost na ISA musí mít VM svojí vlastní virtuální instrukční sadu (V-ISA). Programy se při kompilaci přeloží do V-ISA a pak proběhne jejich distribuce. Na každém fyzickém stroji je pak virtuální stroj implementující danou V-ISA a při interpretaci programu přeloží respektive interpretuje instrukce z V-ISA na instrukce ISA cílového počítače.

V-ISA neobsahuje pouze instrukce ale i množství datových struktur a metadat, samotné instrukce už jsou většinou jednoduché.

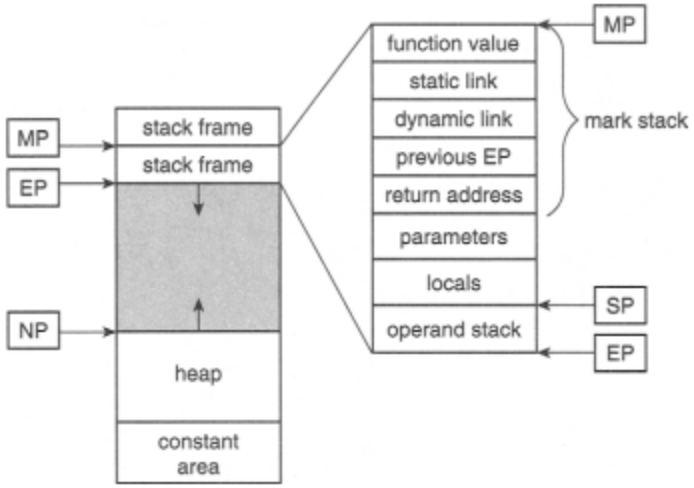
Na následujících stránkách bych rád popsal tři nejvýznamnější virtuální stroje. Prvním je Virtuální stroj pro jazyk Pascal, který byl jakýmsi průkopníkem v této oblasti. Další je pak JVM (Java Virtual Machine) jehož implementaci jsem si sám vyzkoušel a který je v současné době asi nejvýznamnějším virtuálním strojem. Posledním je CLI (Common Language Infrastructure) od Microsoftu. Největší konkurent JVM. JVM a CLI jsou v mnoha ohledech velmi podobné virtuální stroje, proto se u CLI zaměřím především na to v čem se od JVM liší.

Virtuální stroj pro Pascal P-code

P-code VM se zasloužil o rozšíření jazyka Pascal, protože umožnil snadnou přenositelnost kompilátoru Pascalu. Většinu základních konceptů tohoto VM převzali i JVM a CLI. Základní myšlenka je taková: Kompilátor zkompiluje program v Pascalu a vygeneruje program v P-code. P-code je jednoduchá instrukční sada využívající zásobník. Program v P-code už se pak může pustit na P-code VM. Jediné co je potřeba udělat aby program v pascalu mohl běžet všude, je implementace P-code VM pro každou požadovanou platformu a to je mnohem jednodušší než psát celý kompilátor Pascalu.

Druhá velmi důležitá část jsou standardní knihovny, které se starají o komunikaci s hostitelským strojem. Jedná se především o funkce pro čtení a zápis z nebo na standardní výstup a z nebo do souboru. Tyto nativní funkce jsou podporovány přímo virtuálním strojem.

Paměťová architektura P-code V-ISA se skládá z těchto částí: paměť pro instrukce programu, paměť pro konstanty (constant area), zásobník (frame stack) a halda pro alokaci dat. K paměti instrukcí se přistupuje pouze pomocí čítače instrukcí (PC) a je možné jen čtení, nikoliv zápis. Konstanty jsou vygenerovány už překladačem. Zásobník obsahuje rámce. Každý rámec odpovídá jednomu volání funkce. Zásobník pro operandy (operand stack) je součástí rámce. Halda a zásobník rostou v paměti proti sobě a pokud se potkají vyvolá to výjimku.



Rámec se skládá z následujících položek:

- function value - návratová hodnota funkce, pokud něco vrací.
- static link - Pascal umožňuje vnořování funkcí a vnitřní funkce vidí proměnné v jejich rodičích, k jejich nalezení slouží tato položka, která ukazuje na rodičovskou funkci.
- dynamic link - ukazuje na předchozí rámec, neboli hodnota MP pro předchozí rámec.
- previous EP - jak je z názvu zřejmé, hodnota EP pro předchozí rámec.
- return addres - hodnota na kterou by se měl nastavit PC při návratu z funkce

- parameters - parametry funkce
- locals - lokální proměnné
- operand stack - zásobník na operandy, sloužící k vykonávání instrukcí p-code

P-code VM používá tyto registry:

- PC program counter - ukazuje na aktuálně vykonávanou instrukci
- SP stack pointer - ukazuje na vrchol zásobníku operandů
- MP mark stack pointer - ukazuje vrchol zásobníku rámců
- NP new pointer - vrchol haldy, zde se alokuje paměť pro nové proměnné
- EP extreme stack pointer - ukazuje konec aktualního rámce

Maximální velikost zásobníku operandů je známá již za kompilace a tudíž je jasné jak velký rámec maximálně bude, právě na toto místo ukazuje EP, používá se též k hlídání růstu haldy, díky tomu se nemusí kontrolovat kolize z haldou pokaždé když se zvýší SP. SP ukazuje ne aktuální operand na vrchu zásobníku.

Instrukční sada se skládá z instrukcí které mohou přidávat a odebírat operandy z vrcholu zásobníku, matematických instrukcí, logických instrukcí a dalších. Instrukce jsou typované, což znamená že pro sčítání hodnot typu integer je potřeba použít instrukci *adi* ale pro sčítání hondot typu real je instrukce *adr*. Následuje tabulka s příklady některých instrukcí:

Instrukce	Zásobník před	Zásobník po	Popis
adi	i1, i2	i1 + i2	sečete dvě hodnoty typu integer
adr	r1, r2	r1 + r2	sečete dvě hodnoty typu real
dvi	i1 i2	i1 / i2	dělení hodnot typu integer
inn	i1 s1	b1	přítomnost v množině; b1 je pravda pokud i1 je v s1
ldci	i1	i1	načte konstantu typu integer
not	b1	~b1	negace

P-code VM se stal standardem pro ostatní virtuální stroje. Sdílejí spolu především tyto vlastnosti:

- Používá zásobníkově orientovanou instrukční sadu. Díky tomu potřebuje jen minimum registrů a usnadňuje tak interpretaci na jakémkoliv hostitelském stroji. Zásobníková ISA také znamená menší binární programy.
- Paměť je rozdělena do buněk, ale velikost těchto buněk je záležitostí implementace nikoliv ISA.

- Paměť je rozdělena na zásobník a haldu, ale jejich rozsah není dán architekturou. Instrukce nikdy nepoužívají přímé adresy do paměti. Díky tomu je velikost paměti záležitostí implementace.
- Přístup k operačnímu systému je možný jen skrze standardní knihovny. Díky tomu by měl program být na operačním systému nezávislý. Na druhou stranu pak mohou být použity jen sada funkcí, kterou podporují všechny operační systémy.

Virtuální stroje pro vyšší objektově orientované jazyky

Model P-code VM usnadňující přenositelnost se osvědčil a moderní virtuální stroje na něm staví, ale kladou ještě další požadavky na to co by měl virutalní stroj umět. Jedná se hlavně o bezpečnost a ochranu, podporu objektového paradigma, podporu internetu a výkon.

Terminologie není úplně jednoznačná co se týče rozlišení mezi implementací a definicí architektury virtualních strojů. Java Virtual Machine mezi nimi nerozlišuje, termín JVM tak může být použí pro formát ClassFile souborů a instrukcí stejně tak pro konkrétní implementaci na které mohou být spuštěny. Microsoft common language infrastructure je architektura nebo funkční specifikace. Microsoft common language runtime (CLR) je konkrétní implementace vyvinutá Microsoftem. Pokud chceme mluvit jen o instrukční sadě Javy, nazývá se Java bytecode, protože na něj lze pohlížet jako na řetězec bajtů. Instrukční sada CLI se nazývá buď Microsoft intermediate language (MSIL), common intermediate language (CIL) nebo jen IL. Implementaci JVM spolu se standartními Java knihovnami (API) se říká java platforma. CLI implementace se standartními knihovnami se nazývá Microsoft .NET framework.

Požadavek bezpečnosti na virtuální stroj znamená, že program běžící na VM by měl mýt možnost přistupovat jen k jemu povoleným datům a žádným jiným, zároveň by VM měl být chráněn před programem, který vykonává. Běžící program nesmí mít možnost jakkoliv zasahovat do vnitřních struktur virtuálního stroje. Populární jazyky vyšší úrovně jsou silně typované a program musí přesně definovat rozsah v jakém bude přistupovat k datům a jak bude kontrolovat tok programu. Tato informace je předána virtuálnímu stroji skrze V-ISA.

Java program se skládá z binárních souborů které mohou být uloženy lokálně nebo vzdáleném úložišti. Každý takový soubor obsahuje na platformě nezávislý kód a metadata. VM obsahuje takzvaný zavaděč (loader), který binární soubory umí načíst a ověřit že jsou korektní a přeložit je do platformě závislého formátu v paměti virtuálního stroje. V programu mohou být metody uživatelské, systémové a nativní. Uživatelské metody obsahuje uživatelský program. Systémové a nativní jsou součástí standardní knihovny. Loader a standardní knihovna jsou považovány za důvěryhodné. Nativní metody jsou používají především k přístupu k systému. Uživatel také může nadefinovat svojí nativní metodu, ale na vlastní nebezpečí, protože takové metody neprocházejí před spuštěním verifikací.

Další důvěryhodnou částí aplikace je emulátor, který emuluje vykonávání instrukcí V-ISA. Pokud by program chtěl provést nějakou potenciálně nebezpečnou akci třeba chtěl přistoupit

k souboru, standardní knihovna nejdříve požádá o povolení security manager, který rozhodne jestli na takovou akci má program právo.Je důležité ověřit že program nikde neskočí mimo jemu přidělenou oblast paměti a stejně tak čtení a zápis se provede vždy korektně bez možnost zásahu do jiných oblastí. JVM i CLI jsou silně typované a spoléhají ve velkém na kontrolu při zavadění. Díky tomu se za běhu musí provádět jen minimum kontrol.

Podporu objektově orientovaných jazyků tu nechci příliš rozebírat, toto paradigma je tak rozšířené, že předpokládám u čtenáře jeho znalost nebo schopnost si tyto informace snadno doplnit. Přesto zmíním že jde především o podporu objektů jako uživatelem definovaných struktur, dědičnosti, polymorfizmu a skrývání dat.

Garbage collection je další součást moderních VM. Garbage collector (GC) se stará o to aby objekty, které již nejsou dále používány uvolnili místo v paměti pro objekty nové. Není to zadarmo a stojí to určitou část výkonu, ale díky tomu už nemohou nastat chyby, které se vyskytují při manualní správě paměti. Jde o čtení již smazaného objektu nebo o uvolnění paměti, která již jednou uvolněna byla. Jsou, ale i situace, kdy je automatické uvolňování paměti rychlejší než manuální uvolňování paměti pro každý objekt zvlášť.

Protože binární soubory, části programu mohou být umístěny na rozdílných místech na síti, snaží se moderní VM omezit počet dat, které musí být přeneseny. Díky zásobníkové architketuře jsou soubory s kódem menší. Také se využívá lazy zavádění, binární soubory programu jsou načtené až ve chvíli kdy je jich potřeba.

Poslední důležitá vlastnost moderních VM je snaha o co největší výkon. Je to velmi problematické, protože většina předchozích vlastností jde proti tomuto požadavku. Avšak existuje mnoho možností jak výkon zlepšit. Jako jsou úpravy instrukční sady, překlad částí kódu do nativního jazyka a další pokročilejší metody.

Java Virtual Machine

Datové typy

Specifikace JVM stejně jako jazyk Java dělí datové typy na dvě skupiny: primitivní datové typy a reference. Reference může ukazovat na objekt nebo pole. Objekt se skládá a pole obsahuje buď primitivní datové typy nebo reference. Primitivní datové typy v JVM jsou: byte, short, int, long, char, float, double, boolean a returnAddress. Tyto typy nejsou definované počtem bitů, který zabírají ale rozsahem hodnot, kterého mohou nabývat. To umožňuje implementaci jak je bude ukládat. Referencí jsou tři druhy: reference na třídu, reference na rozhraní a reference na pole. Reference buďto ukazuje na nějaký objekt a nebo má hodnotu null. Jak přesně má být reference zakódována není určeno.

Oblasti s daty

PC

Stejně jakou u P-code VM ukazuje na právě vykonávanou instrukci.

JVM zásobník (Stack)

Zásobník který obsahuje rámce. K zásobníku není přestupováno přímo a proto mohou být rámce uloženy i na haldě. Rámec je na zásobník přidaný v okamžiku kdy je volaná funkce a v okamžiku návratu je z něj rámec zase odebrán. Návratová hodnota je vložena na zásobník operandů předchozího rámce.

Halda (Heap)

Zde jsou alokovány všechny instance tříd a pole. Halda si musí automaticky spravovat a uvolňovat paměť, neboli musí být použit nějaký garbage collector. Není specifikováno jaký typ Gc má být použit ani jak budou data přesně uložena v paměti.

Oblast metod (Method Area)

Navzdory svému jménu jsou zde uložena data podle java tříd. Může opět být součástí haldy, ale nemusí. Ke každé třídě jsou zde uloženy veškeré informace včetně byte kódů jejich metod. Třídy se načítají až ve chvíli kdy jsou potřeba.

Constant pool

Obsahuje konstanty pro třídu nebo rozhraní. Jsou to jak primitivní typy, tak i třeba hodnoty typu String.

Rámec

Nová rámec je vložen na zásobník pokaždé když je volána metoda a je ze zásobníku odebrán když vykonávání metody skončí. Každý rámec má svoje vlastní pole s lokálními proměnnými, vlastní zásobník operandů a referenci na identifikátor třídy (obsahující aktuální metodu) ukazující na constant pool. Velikost zásobníku operandů a počet proměnných v constant poolu jsou vypočítány při kompilaci.

Lokální proměnné

Není dáno kolik musí jedna lokální proměna zabírat bajtů, ale představuje jakousi buňku do které se musí vejít všechny datové typy, kromě double a long. Ty se musí vejít do dvou takových buněk.

Zásobník operandů

Je jedinečný pro každý rámec a po vytvoření rámce je prázdný. Je to paměť s kterou operuje většina instrukcí. Protože je JVM zásobníkový počítač, berou si instrukce operandy většinou přímo z tohoto zásobníku. Existuje tak řada instrukcí, které na něj umí přidat lokální proměnnou nebo konstantu. A instrukce pro sčítání nebo pro volání dalších funkcí očekávají, že pro ně budou na zásobníku přichystané parametry.

Do každé buňky zásobníku operandů se musí vejít kterýkoliv podporovaný typ včetně typů long a double. Instrukce jsou typovány a není možné například vložit dvakrát int na zásobník a pak na zavolat instrukci pro long.

Instrukční sada JVM

Istrukce JVM je opcode velikosti jednoho bajtu a za ním následuje nula nebo více operandů. Tyto operandy představují argumenty nebo data pro instrukci. JVM pracuje dokud jí nedojdou instrukce v bajtkódu aktualní metody. Nejprve spočte novou hodnotu PC a načte opcode instrukce na pozici dané PC. Pokud daná instrukce má další operandy. JVM je načte. Poté je instrukce vykonána. Pokud JVM nedošla na konec bajtkodu celý cyklus se opakuje od začátku. V opačném případě byla provedena celá metoda.

Drtivá většina instrukcí je typovaná a může pracovat jen s daným typem. Pro typy boolean, char, byte a short však platí že na zásobníku operandů se rozšíří na int. Díky tomu se počet instrukcí drží na rozumné úrovni. Při ukládání do paměti jsou tyto typy přetypovány na požadovanou velikost. Pole bajtů tak pořád lze uložit o poznání úsporněji než pole hodnot typu int.

Instrukce pro načítání a ukládání dat

Tyto instrukce slouží pro přenos dat mezi lokálními proměnnými a zásobníkem operandů uvnitř rámce. Lze je rozdělit do několika skupin:

- Vložení lokální proměnné na zásobník operandů
- Uložení hodnoty ze zásobníku operandů do lokální proměnné
- Vložení konstanty na zásobník operandů
- Získáním přístupu k více lokálním proměnným pomocí rozšíření indexu

Kromě těchto instrukcí mohou přidávat a odebírat hodnoty z zásobníku operandů i instrukce manipulující s proměnnými objektu a poly.

Aritmetické instrukce

Aritmetické instrukce pracují nad dvěma typy dat. Nad celými čísly a nad čísly s pohyblivou řádovou čárkou. Typicky je výsledek funkcí dvou parametrů. Parametry jsou načteny ze zásobníku operandů a výsledek je tam pak vložen. Jedná se o tyto skupiny instrukcí

- Součet
- Rozdíl
- Násobenek
- Podíl
- Zbytek po dělení
- Negace
- Bitový posun
- Bitová disjunkce
- Bitová konjunkce
- Bitová exkluzivní disjunkce
- Inkrementace lokální proměnné

- Porovnání

JVM nevyhazuje výjimku v případě přetečení nebo podtečení při výpočtu.

Instrukce pro přetypování

Tyto instrukce slouží k přetypování všech číselných typů mezi sebou nejjednodušeji je lze rozdělit na tyto dvě třídy:

- Rozšiřující konverze
- Zužující konverze

Při obou konverzích může dojít ke ztrátě nějaké informace jako je ztráta přesnosti nebo oříznutí nejnižších bitů. JVM však v těchto případech nevytváří žádné výjimky.

Instrukce pro manipulaci s objekty

Instance tříd i pole jsou objekty, ale JVM k manipulaci s nimi používá odlišné instrukce.

- Vytvoření nového pole
- Vytvoření nové instance třídy
- Přístup k instančníma třídním proměnným
- Vložení prvku pole na zásobník operandů
- Uložení hodnoty ze zásobníku operandů do pole
- Vložení délky pole na zásobník operandů
- Kontrola zda je objekt nějakého typu nebo potomkem

Instrukce pro práci se zásobníkem

Tyto instrukce manipulují s položkami zásobníku bez ohledu na typ. Jedná se o instrukce pro smazání hodnoty z vrchu zásobníku, duplikace hodnoty na vrchu nebo prohození dvou nejvyšších hodnot na zásobníku.

Instrukce ovlivňující tok programu

Tyto instrukce buď vždy nebo pod nějakou podmínkou mění tok programu a tedy hodnotu PC, jinak řečeno další instrukci která se načte.

- Podmíněný skok
- Složený podmíněný skok
- Nepodmíněný skok

Instrukce pro volání metod a návrat z metod

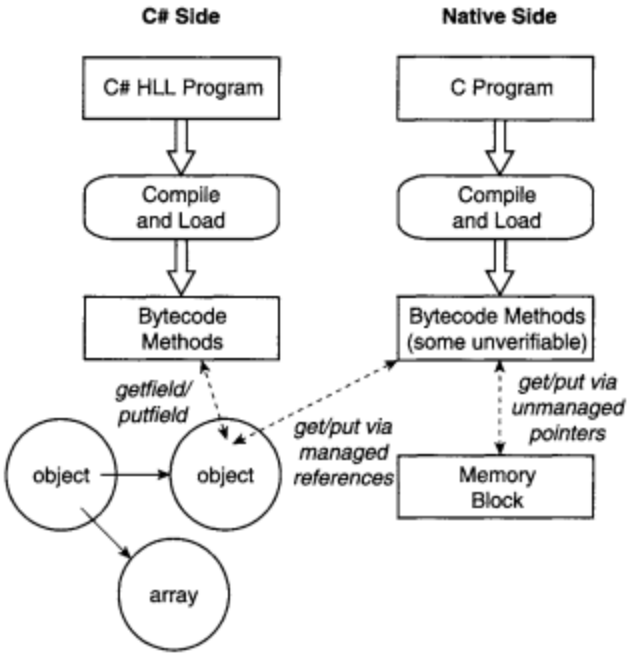
- invokevirtual - zavolá metodu objektu, metoda se hledá podle typu objektu
- *invokeinterface* - volá metodu rozhraní, hledá implementaci metody v objektu
- invokespecial - volá speciální instanční metodu, například konstruktor nebo privátní metodu
- invokedynamic - instrukce není určená pro Javu, ale pro ostatní jazyky, umožňuje snadné použití ducky typingu

Common Language Infrastructure

CLI podporuje objektově orientované výpočty stejně jako JVM, ale cíle tohoto virtuálního stroje jsou zdánlivě širší než ty Javy. Oba stroje jsou si velmi podobné v tom že jejich běhové prostředí provádí kontroly programů a garbage collection. JVM byla navržena přímo pro jazyk Java, ale řada jazyku jde zkompilovat do binárních java souborů. Oproti tomu je CLI navržena, tak aby podporovala množství různých spolupracujících jazyků. Dokonce na ní mohou běžet i programy které nejsou nebo nemohou být validovány zavaděčem. Hlavním rozdílem mezi JVM a CLI tedy je, že CLI je nezávislá nejen na hostitelském systému, jako Java, ale též na jazyku vyšší úrovně.

Obdobou binárních Java souborů jsou na platformě .NET moduly. Také obsahují jak metadata tak kód ve formě MSIL (obdoba Java bajtkódu). Do těchto modulu lze zkompilovat programy v řadě jazyků. Některé z nich jsou: Java, C#, Visual Basic, .NET a C++. Moduly jsou při načítání validovány stejně jako v případě JVM, na rozdíl od něj ale lze tuto validaci vypnout.

V JVM lze volat nativní metody a nativní kód se může dotazovat na různá data pomocí funkcí JVM. V CLI je spolupráce dotažena ještě dál, lze sdílet i datové struktury mezi různými jazyky. Má to však svou cenu. Může to vyžadovat velké změny implementace současných jazyků. Jak se stalo třeba v případě Visual Basicu a dochází i k dalším problémům, třeba když je jeden jazyk citlivý na velikost znaků a druhý nikoliv.



volání nativního kódu v CLI

Instrukční sada MSIL, je bajtkódu velmi podobná. Kvůli podpoře jazyku bez verifikace však obsahuje řadu nekontrolovaných instrukcí pro alokování kusu paměti a přístup k nějaké její části. Nebo pro práci s ukazateli. Instrukce pro aritmetiku nejsou typované a CLI pozná až za běhu jak instrukci provést podle toho co se nalézá na zásobníku operandů.

JVM & CLI pojmy

	JVM	**CLI**
architektura VM	JVM (specifikace)	CLI
nejznámější implementace VM	JVM (implementace)	CLR
platforma (VM + standartní knihovny)	Java platform	.NET framework.
instrukční sada	Java bytecode	MSIL
binarní spustitelné soubory (bytecode + metadata)	Java class file	Modul

Slovík pojmů

ISA	Instruction Set Architecture	instrukční sada
I/O	input/output	vstup/výstup
VM	virtual machine	virtuální stroj
ABI	application binary interface	
API	application programming interface	
HLL VM	High Level Language Virtual Machine	Virtuální stroj pro vyšší jazyk
JVM	Java Virtual Machine	
CLI	Common Language Infrastructure	
P-code		
loader		zavaděč
bytecode		bajtkód

Zdroje

SMITH, James a Ravi NAIR. *Virtual machines: versatile platforms for systems and processes*. Amsterdam: Elsevier, 2005, 638 s. ISBN 15-586-0910-5.

LINDHOLM .., Tim.. *The Java virtual machine specification*. Java SE 7 ed. Upper Saddle River, NJ: Addison-Wesley, 2013. ISBN 01-332-6044-5.

Pascal Implementation. [online]. [cit. 2015-01-19]. Dostupné z: http://homepages.cwi.nl/~steven/pascal/book/pascalimplementation.html

Difference between api and abi. [online]. [cit. 2014-11-30]. Dostupné z: http://stackoverflow.com/questions/3784389/difference-between-api-and-abi

API. [online]. [cit. 2014-11-30]. Dostupné z: http://cs.wikipedia.org/wiki/API

Virtual machine. [online]. [cit. 2014-11-30]. Dostupné z: http://en.wikipedia.org/wiki/Virtual_machine

P-code machine. [online]. [cit. 2015-01-19]. Dostupné z: http://en.wikipedia.org/wiki/P-code_machine