Metody přidělování paměti

Základní způsoby: -Statické (přidělění paměti v čase překladu)
-Dynamické (přiděleno v run time) — **v zásobníku** / **na haldě**

Důležitá hlediska jazykových konstrukcí:

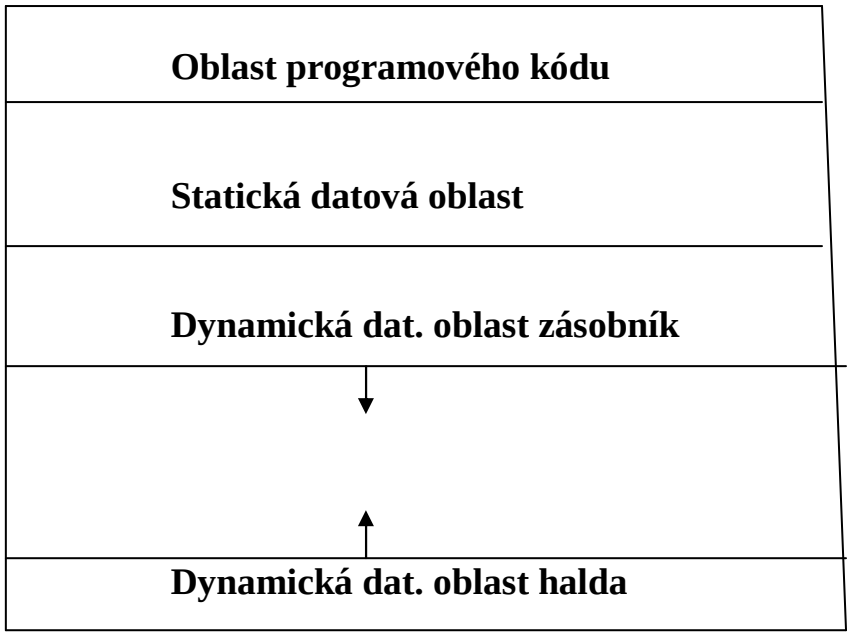
- **Dynamické typy**
- **Dynamické proměnné**
- **Rekurze**
- **Konstrukce pro paralelní výpočty**

Podstatný je rovněž způsob:

- **Omezování existence entit v programu (namespace, package, blok…)**
- **Vnořování a určování přístupu k nelokálním entitám**
  na základě statického vnořování rozsahových jednotek,
  na základě dynamického vnoření rozsahových jednotek.

Vnořování podprogramů podporují např.: Algol, Pascal, Lisp, GCC, Fortran, Matlab, JavaScript,...
Vnořování podprogramů nepodporuje C a bez přímé podpory jsou: C++, C#, Java. Dovolují docílit stejný efekt způsoby: Definicí tříd uvnitř tříd, funkčními objekty, vnořovanými lambda výrazy, anonymními funkcemi

Rozdělění paměti cílového programu



Statické přidělování paměti lze použít pro:

- **Globální proměnné**
- **Static proměnné**
- **Proměnné jazyka bez rekurze (i s vnořenou blokovou strukturou)**

Př.

Blok1
místoA
Blok2
místoB
Blok3
místoC
Konec bloku3

Blok4
místoD
Konec bloku4
místoE
Konec bloku2
místoF
Konec bloku1

Statické přidělování lze realizovat pomocí zásobníku

Ukažme obsah zásobníku v různých okamžicích výpočtu

Bloky

		3	4		
	2	2	2	2	
1	1	1	1	1	1
Místo A	**B**	**C**	**D**	**E**	**F**

-Vnořování podprogramů (funkcí, metod) je ale složitější, viz dále.

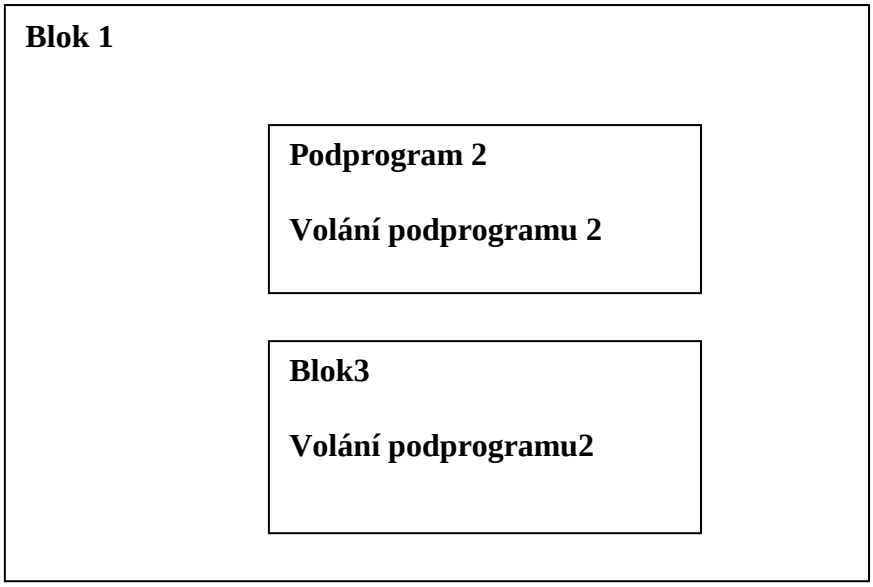
-Pokud jazyk nedovoluje vnořování (př. C), pak jsou přístupné jen globální proměnné a proměnné z vrcholového AZ v zásobníku

Dynamické přidělování v zásobníku

Část paměti přidělovaná při vstupu výpočtu do rozsahové jednotky programu se nazývá Aktivační Záznam (AZ představuje lokální prostředí výpočtu). Obsahuje místo pro:

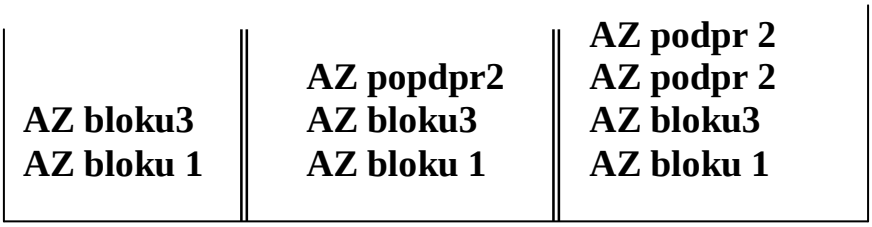
- **Lokální proměnné**
- **Parametry (je-li rozsahovou jednotkou podprogram či funkce)**
- **Návratovou adresu („ „)**
- **Funkční hodnotu (je-li rozsahová jednotka funkcí)**
- **Pomocné proměnné pro mezivýsledky (také možno v registrech)**
- **Další informace potřebné k uspořádání aktivačních záznamů**

Př.1 Vnořování rozsahových jednotek



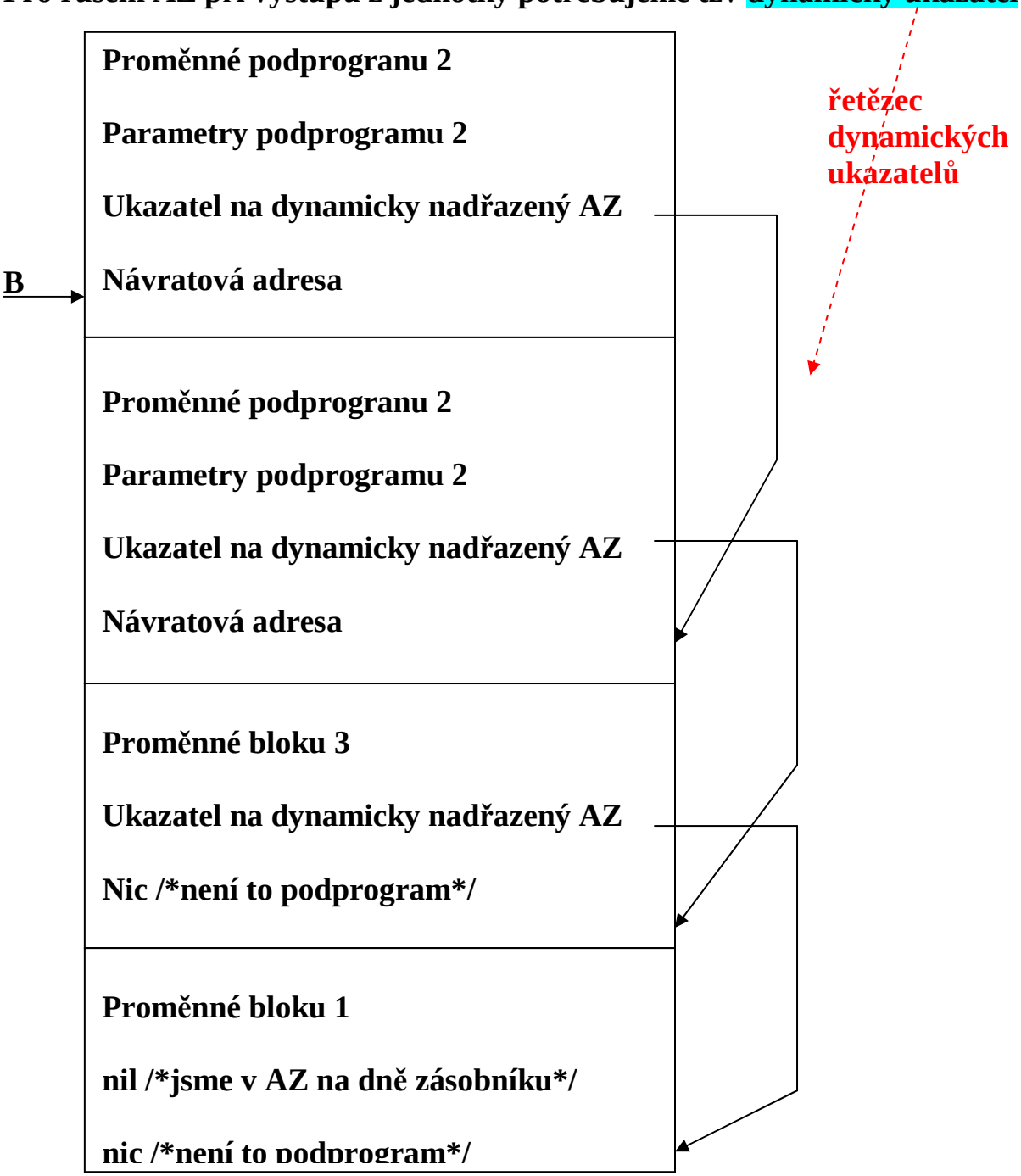
? stav výpočtového zásobníku v různých časech výpočtu

a) **Při vstupu do bloku 3**
b) **Při prvním volání podprogramu 2**
c) **Při rekurzivním vyvolání podprogramu 2**



a) b) c)

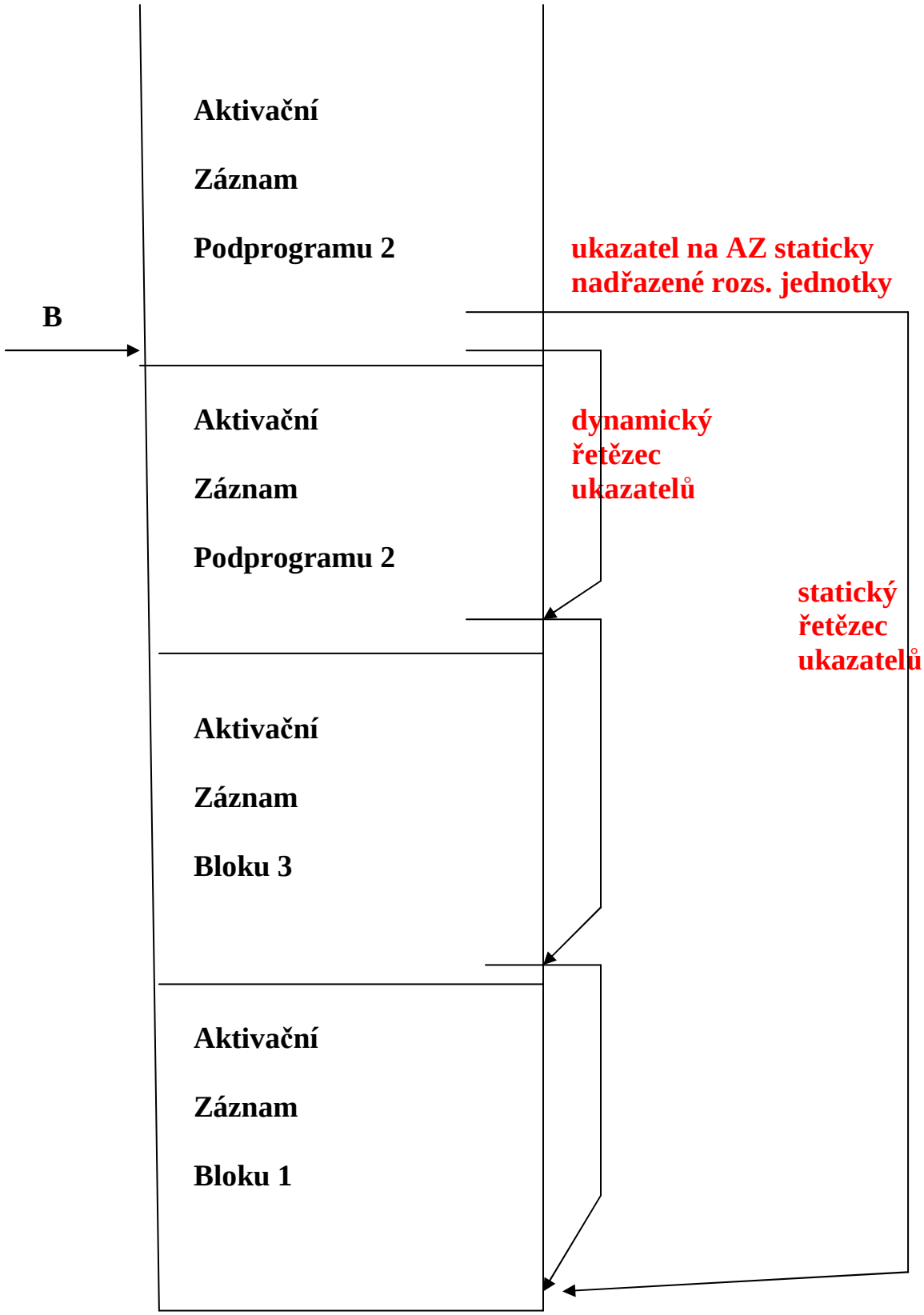
Jaké bude uspořádání aktivačních záznamů při rekurz. vyvolání podpr.2 ?
Pro rušení AZ při výstupu z jednotky potřebujeme tzv dynamický ukazatel



Obr. Zásobník při rekurzivním volání podprogramu 2

B je registr ukazující na vrcholový AZ

Potřebujeme ještě vyřešit přístup k nelokálním proměnným při statickém = lexikálním rozsahu platnosti jmen. To řeší tzv. řetězec statických ukazatelů

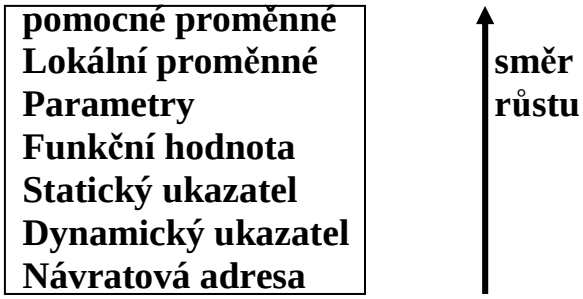


Obr. Zásobník se statickým (ukazuje na lexikálně nadřazený AZ) a dynamickým řetězcem ukazatelů při rekurzivním volání podprogramu 2

Pozn.: Statický uk. je nakreslen (pro přehlednost) jen u AZ rek. volání podpr.2

Vytváření řetězců ukazatelů

Nechť AZ má tvar:



Uvažujme zásobník Z, s vrcholem (nejvyšší zabranou adresou) T,
n je hladina deklarace , m je hladina volání
Při vstupu do rozsahové jednotky (vyvolání podprogramu nebo vstupu výpočtu do bloku = Aktivace rozsahové jednotky):

A1) Z[T + 1] ← návratová adresa /* pouze u podprogramů*/
A2) Z[T + 2] ← B /*nastavení dynamického ukazatele*/
A3) Z[T + 3] ← B
For i ← 1 to m – n do Z[T + 3] ← Z[Z[T + 3] + 2] /*nastavení statického ukazatele*/
A4) B ← T + 1 /*nastavení bázového registru*/
A5) T ← T + velikost aktivačního záznamu
A6) skok na první instrukci podprogramu a uložení do Z údajů o skutečných parametrech /*pouze u podprogramů*/

Pozn. Je-li podprogram překládán odděleně (neznámá velikost jeho AZ), pak je úprava T provedena až na začátku volaného podprogramu.

Při výstupu z rozsahové jednotky (Návrat z podprogramu nebo průchod koncem bloku):

N1) T ← B – 1
N2) B ← Z[B + 1]
N3) skok na adresu uloženou v Z[T + 1] /*pouze u podprogramů*/

Výstup z rozsahové jednotky nelokálním skokem (hladina n deklarace návěští je menší než hladina m místa s příkazem skoku)
Vždy platí n ≤ m

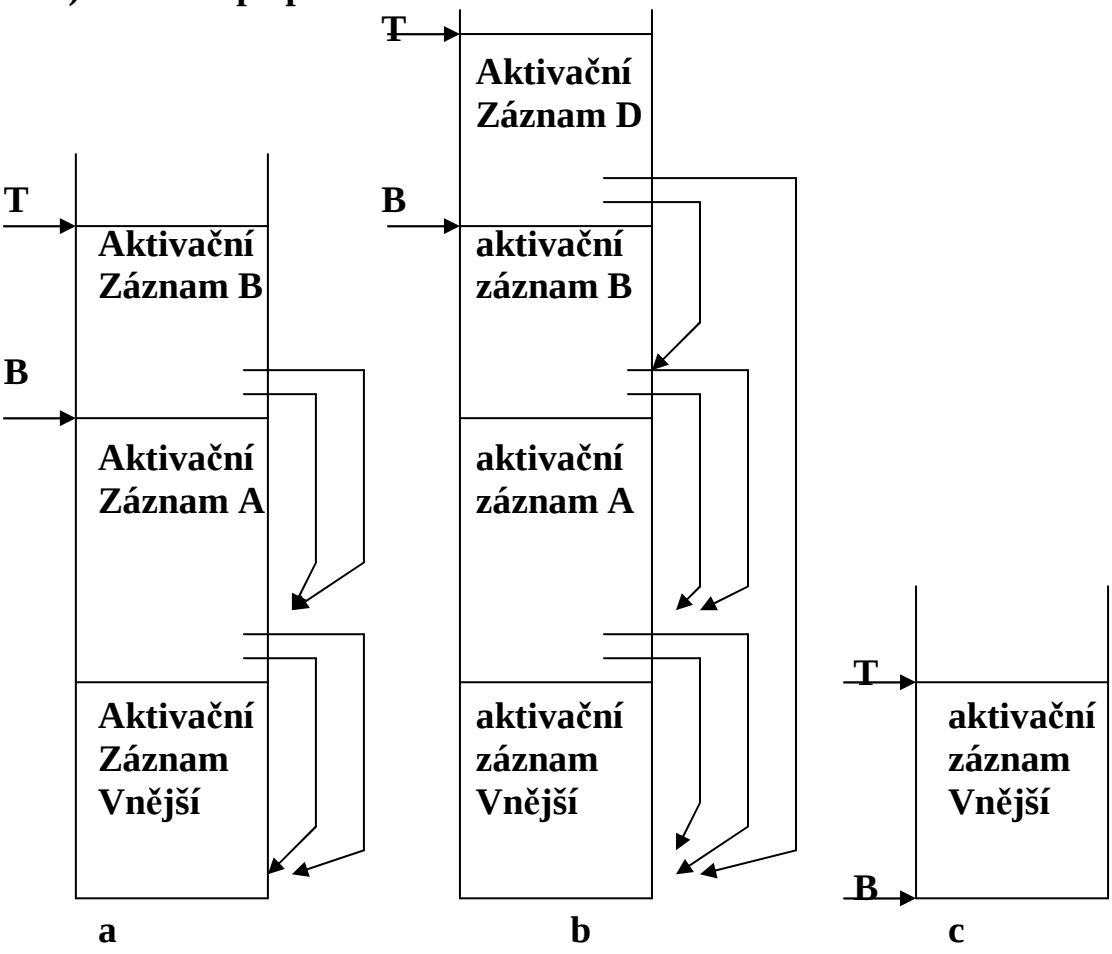
```
S1) for i ← 1 to m – n do { Pom ← B
                            repeat   T ← B – 1
                                     B ← Z[ B + 1]
                            until B ≠ Z[ POM + 2]
                          }
```

S2) skok na adresu, kterou návěští představuje

Př.2

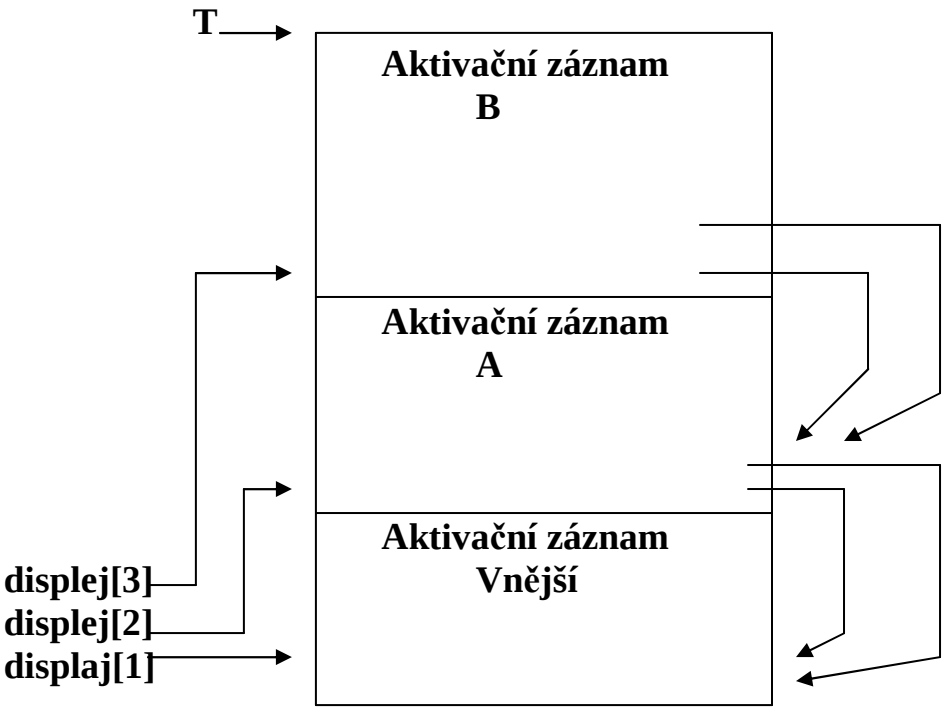
```
Podprogram Vnější //označme n hladinu deklarace
Label 1                    // a m hladinu místa použití
        Podprogram A
                Podprogram B
                Volání D  //zde  n=1; m=3
        Volání B
        Podprogram D
        Goto 1        //zde n=1; m=2
Volání A
1:
```

a) obsah Z při provádění B, před voláním D,
b) obsah Z po vyvolání D, ped provedením nelokálního skoku,
c) obsah Z po provedení skoku.

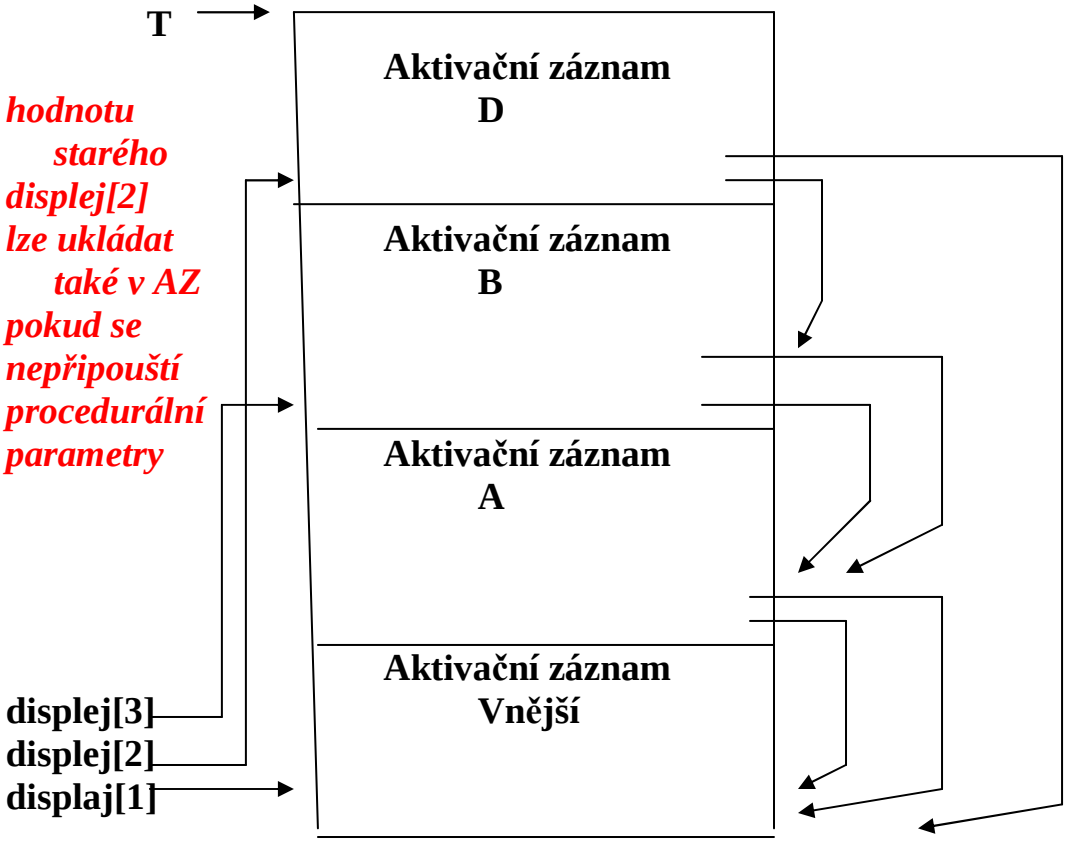


ad b) je to stav v okamžiku volání podpr. s hladinou deklarace 1, volaného v místě s hladinou 3
ad c) stav po výskoku z hladiny 2 do místa s hladinou 1

Zrychlení přístupu k nelokálním proměnným
(pomocí vektoru ukazatelů displej[i], kde i je hladina rozs. jedn.)



Obr. Stav Z při výpočtu B z př.2



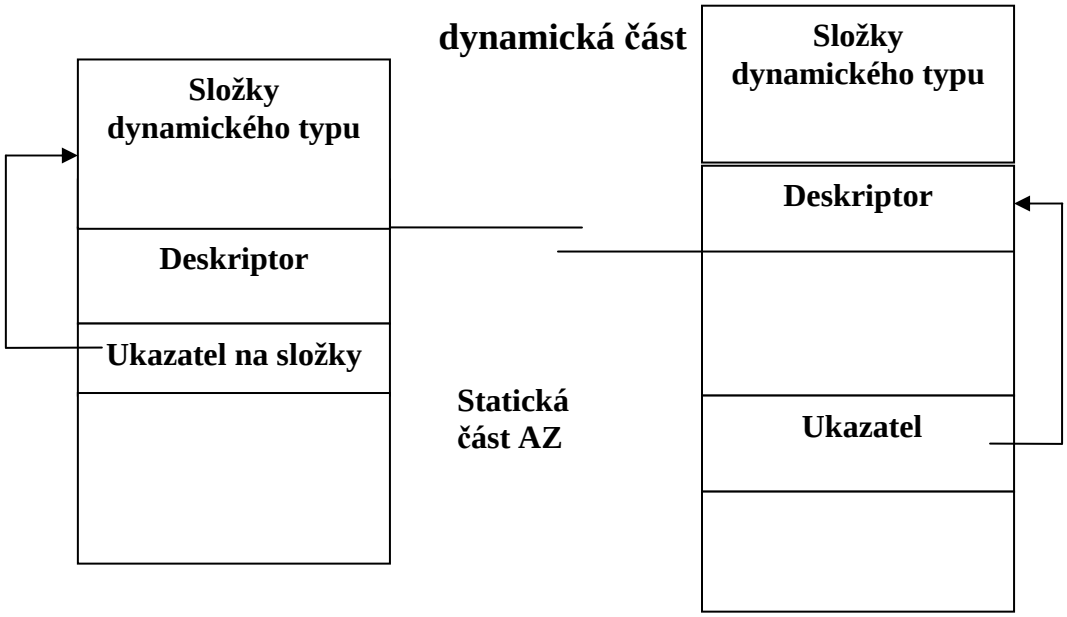
hodnotu starého displej[2] lze ukládat také v AZ pokud se nepřipouští procedurální parametry

Obr. Stav Z při výpočtu D z př.2

Dynamická adresa proměnné je dvojice (n, p) = displej[n] + p

Dynamické datové typy (typicky pole s proměnnými mezemi)

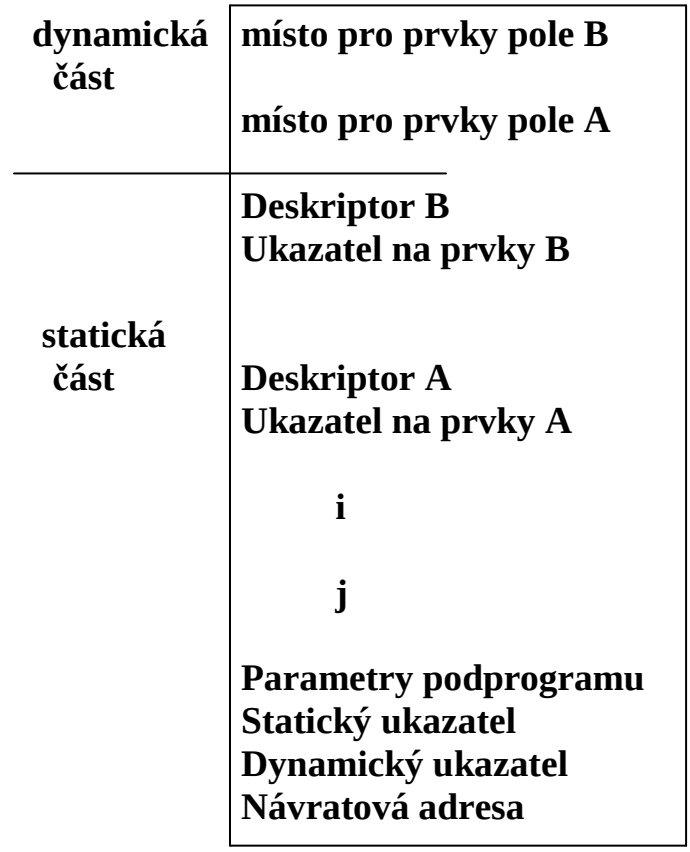
Možnosti struktury aktivačního záznamu pro dynamický typ



Deskriptor se vytvoří při překladu, uchovat se ale musí i při výpočtu

Př.3 Aktivačního záznamu s dynamickými typy

```
podprogram PRIKLAD;
        int i, j ;
        int A(m .. n);
        int B(p .. q, r .. s,  );
```



Předávání parametrů podprogramům

-hodnotou (C, C++, Java,C#) formální parametr je lokální proměnnou do níž se předá hodnota

-odkazem (C, C++ je-li parametrem pointer, objektové parametry Javy, C# označené ref) předá informaci o umístění skutečného parametru

-výsledkem - formální parametr je lokální proměnnou z níž se předá hodnota do skutečného parametru před návratem z podprogramu slouží jako výstupní parametr

-hodnotou výsledkem (novější Fortran) - kombinace

-jménem – má efekt textové substituce (jako historická zajímavost)

-v případě strukturovaných parametrů

-jsou-li to statické typy ⇒ předá se adresa prvého prvku

-jsou=li to dynamické typy ⇒ předá se ukazatel na descriptor

-je-li parametrem podprogram

-u jazyků nedovolujících hnízdění podprogramů ⇒předá se adresa začátku = pointer

- u jazyků dovolujících hnízdění podprogramů ⇒

spolu s adresou musí předdat i platné prostředí. Jsou různé možnosti co považovat za platné prostředí:

-**mělká vazba** ⇒ platné je prostředí v němž se nachází volání formálního podprogramu

-**hluboká vazba** ⇒platné je prostředí kde je předávaný podprogram definován

-**ad hoc vazba** ⇒platné je prostředí kde je vydán příkaz volání podprogramu jež má za parametr podprogram

Př.4

```
Podprogram  P1( )  {
        Prom x ;
        Podprogram P2 ( )  {
                Vytiskni (x) ; /*co se tady tiskne?*/
        };
        Podprogram P3 ( )  {
                Prom x ;
                x ← 3;
                P4(P2) ;
        };
        Podprgram P4( podprogram Px )  {
                Prom x ;
                x ← 4;
                call Px( );
        }
        x←1;
        P3( ) ;
}
```

prostředí, kde je předávaný podprogram definován

prostředí, kde je vydán příkaz volání s parametrem podprog.

prostředí, v němž je volán formální podprogram

Při mělké vazbě se tiskne … ?

Při hluboké vazbě se tiskne … ?

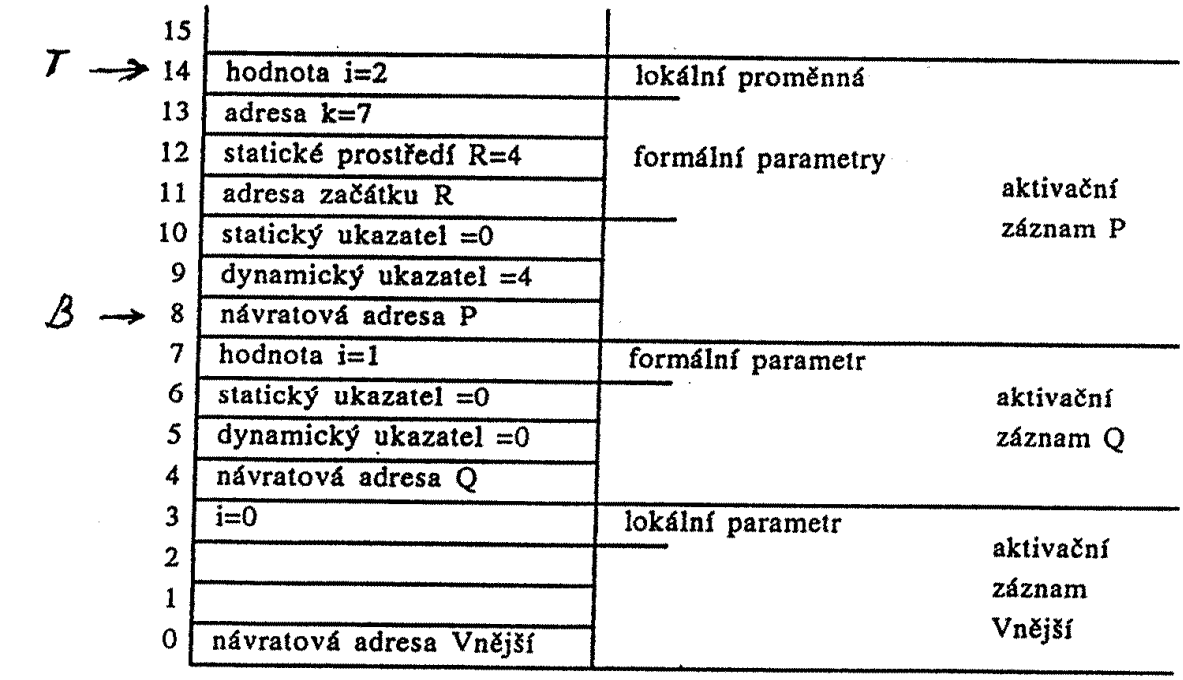
Při ad hoc vazbě se tiskne … ?

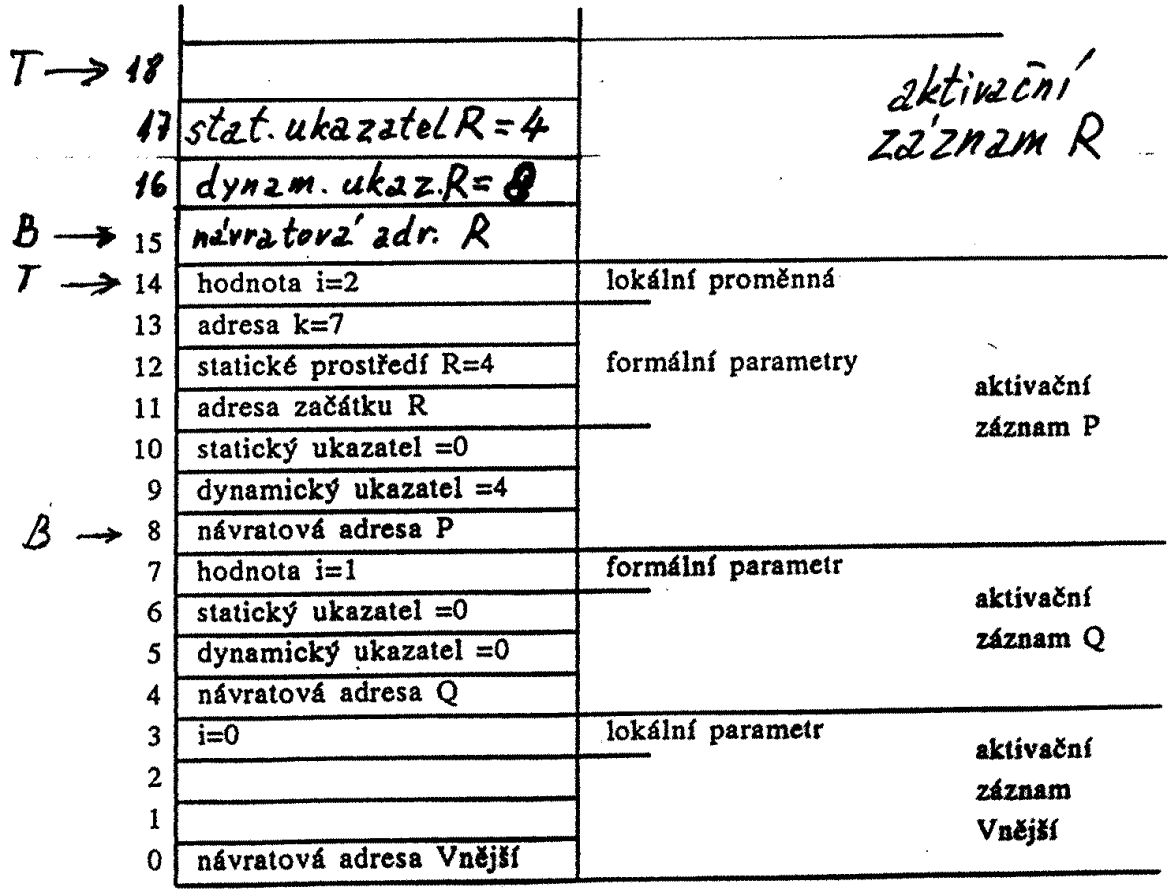
Př.5

Předpokládejme hlubokou vazbu. Co se vytiskne po spuštění procedury Vnější?

```
podprogram Vnejsi;  {
        prom i:int;
        podprogram P( podprogram FP; prom k:int;)  {
                prom i:int;
                i←k+1;  FP( ); tisk(i);
        }
        podprogram Q(i:int); {
                podprogram R ( )  {
                        Tisk(i);
                }
                P(R,i);
        }
        i← 0; Q(i+1);
}
```

Stav před vyvoláním a po vyvolání formálního poprogramu FP z př.5





Přidělování paměti pro paralelní výpočty

Pro uložení AZ paralelního výpočtu nutno použít haldu nebo zobecněný zásobník

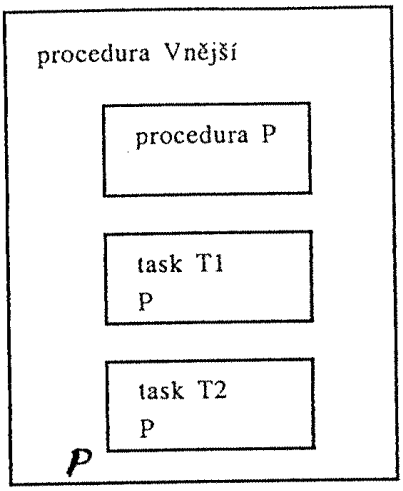
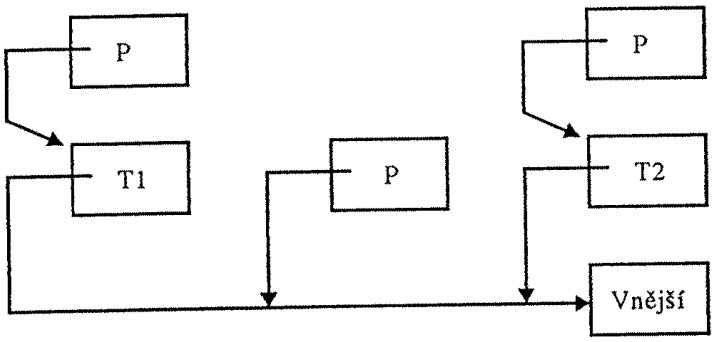
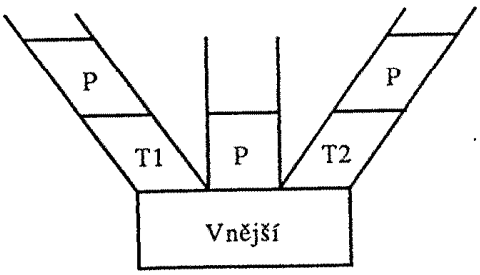


Schéma vnoření bloků programu

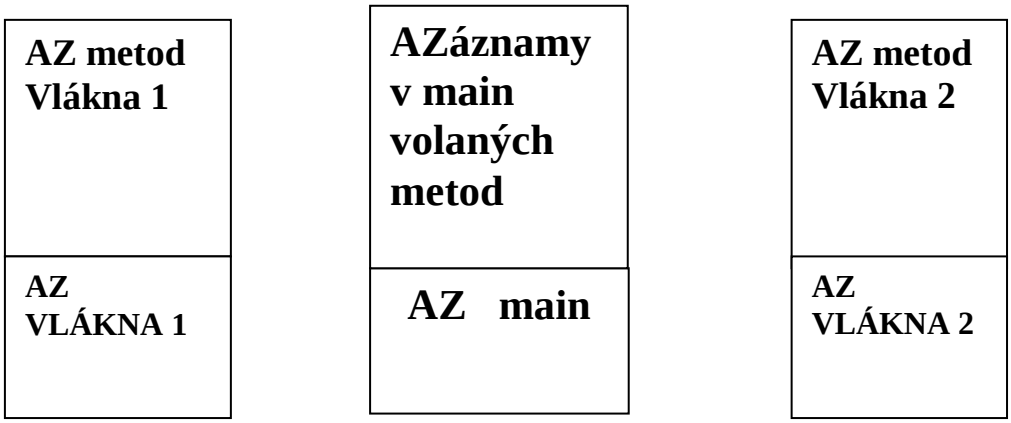


Struktura aktivačních záznamů při paralelním výpočtu

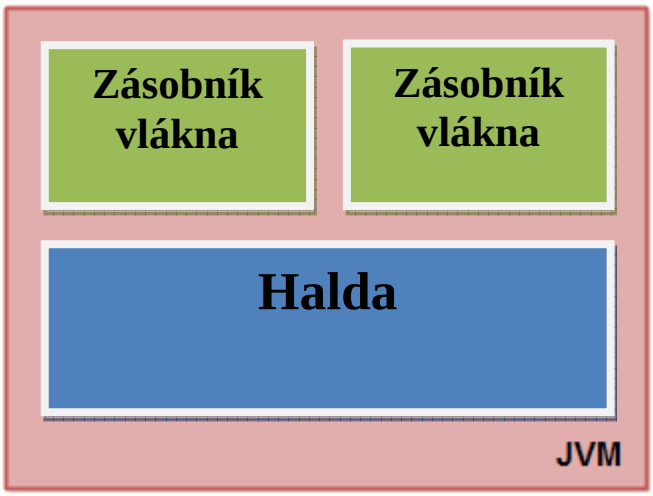


Uložení aktivačních záznamů ve zobecněném zásobníku

Př v javovském prostředí



Paměťový model Javy

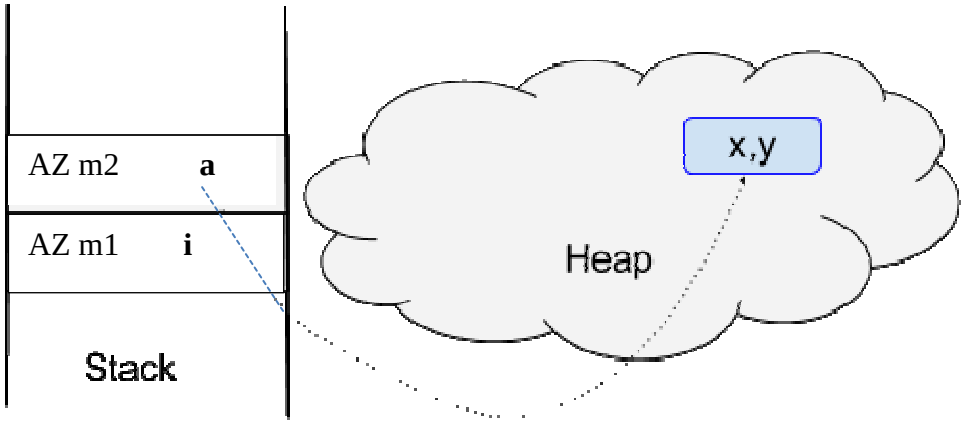


- Halda obsahuje všechny objekty vytvořené v aplikaci, bez ohledu ve kterém to bylo vláknu.
- Každé vlákno v JVM má svůj vlastní zásobník obsahující
  - Informaci o metodách vyvolaných vláknem
  - Lokální proměnné
  - Lokální proměnné primitivních typů
  - Lokální proměnné referenčního typu.

Paměťový model Javy

Př.

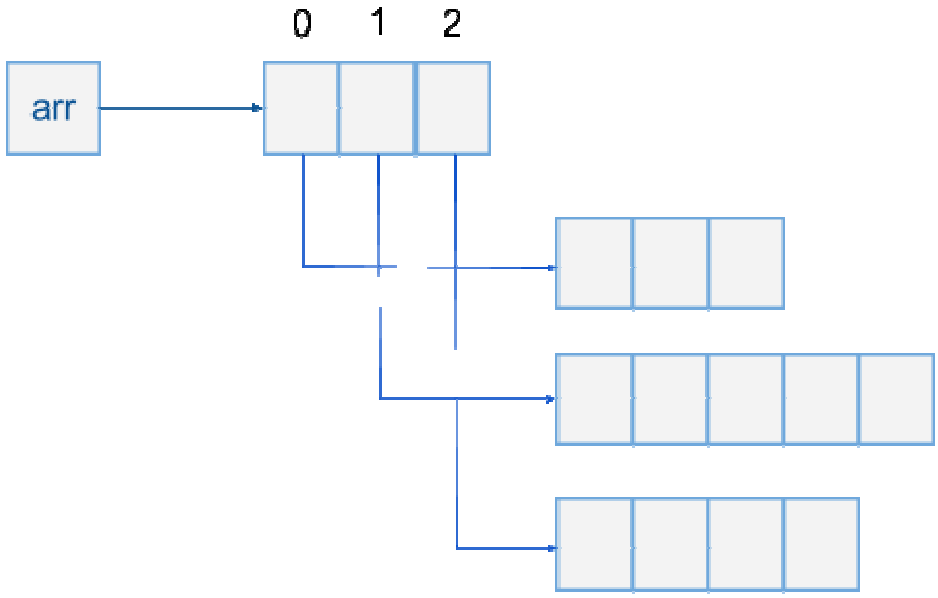
```
class A {
        int x;
        int y;
}

...

public void m1() {
        int i = 0;
        m2();
}

public void m2() {
        A a = new A();
}

...
```

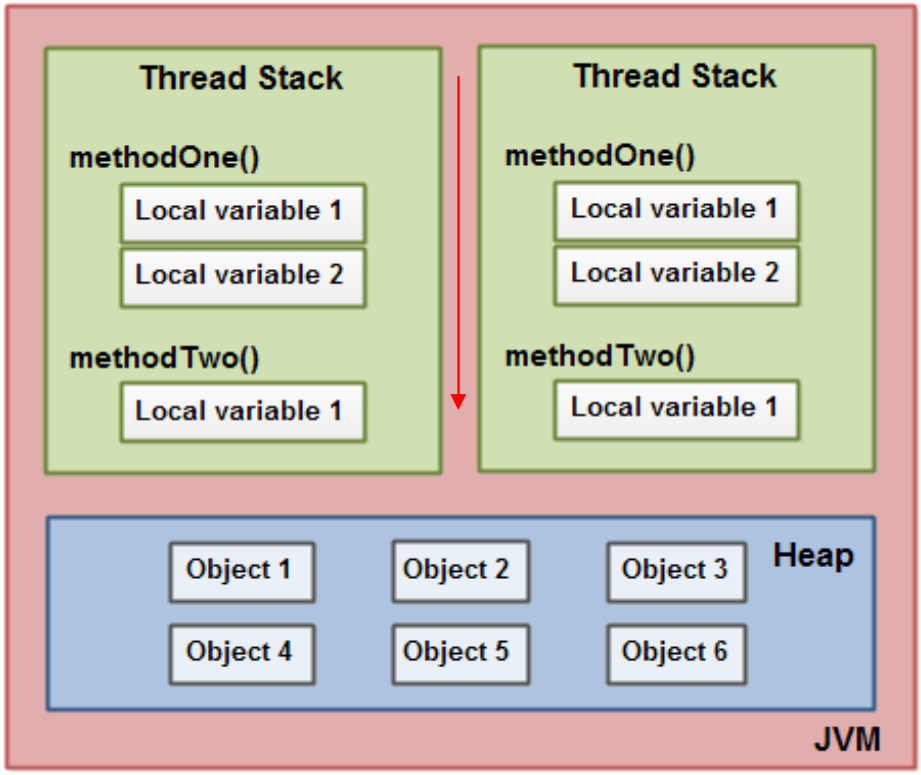


Pole Javy

Pole v Javě jsou objekty – na haldě

```
int[ ][ ] arr = new int[3][ ];
arr[0] = new int[3];
arr[1] = new int[5];
arr[2] = new int[4];
```

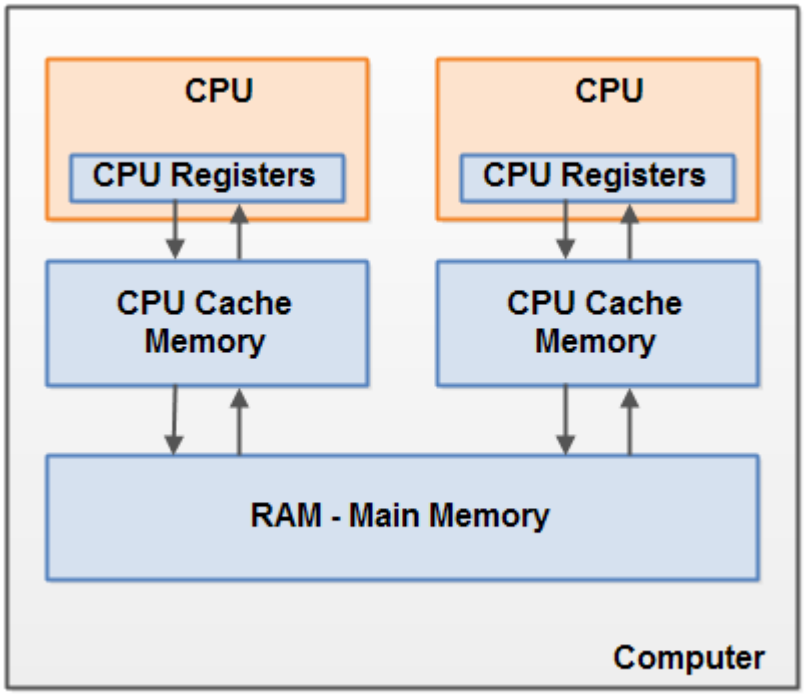




Obr. Příklad obsazení zásobníku a haldy

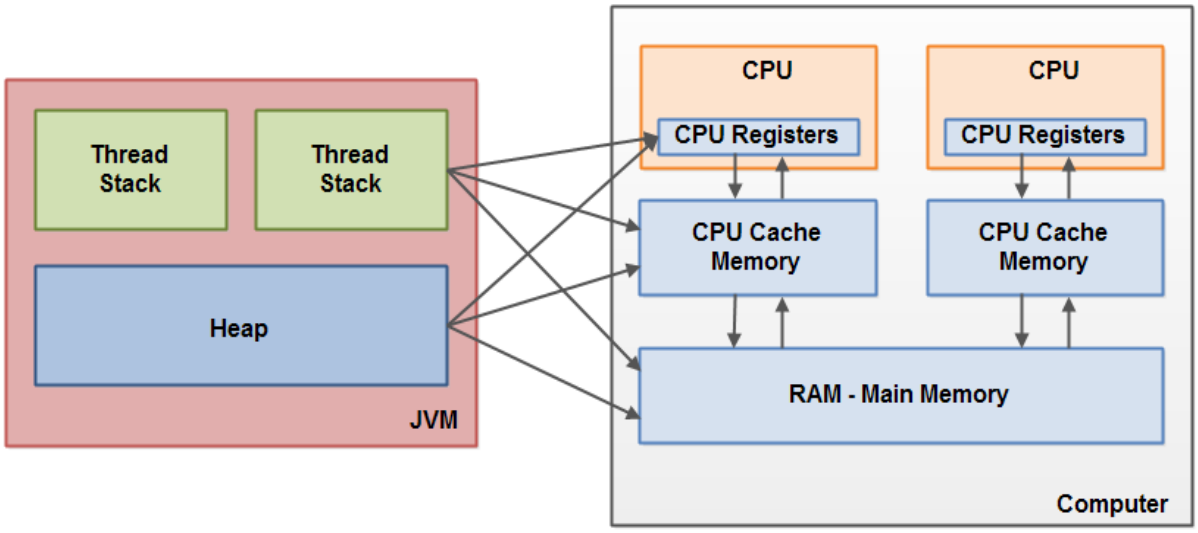
- ✓ Členské proměnné objektů jsou umístěné na haldě
- ✓ Proměnné statických tříd spolu s definicí tříd jsou také na haldě.
- ✓ Objekty na haldě jsou zpřístupněny všem vláknům, která mají přístup k objektu.
- ✓ Když má vlákno přístup k objektu, může také přistupovat k členským proměnným objektu.
- ✓ Volají-li dvě vlákna metodu na stejném objektu současně, mají obě přístup ke členským proměnným objektu. Každé vlákno ale bude mít svou vlastní kopii lokálních proměnných.

Hardwarová paměťová architektura



- Obvykle 2 nebo více CPU
- Simultánní běh vláken
  - CPU registry – nejrychlejší přístup
  - Cache registry – středně rychlé
  - Vlastní vnitřní paměť – relativně nejpomalejší

- Hardwarová architektura nerozlišuje mezi zásobníky vláken a haldou, vše je v paměti
- Části zásobníku i haldy mohou být dočasně v registrech CPU nebo v cache



problémy:

a) viditelnost sdílených proměnných pro vlákna po update
b) Podmínky soutěže o přístup při sdílení proměnných

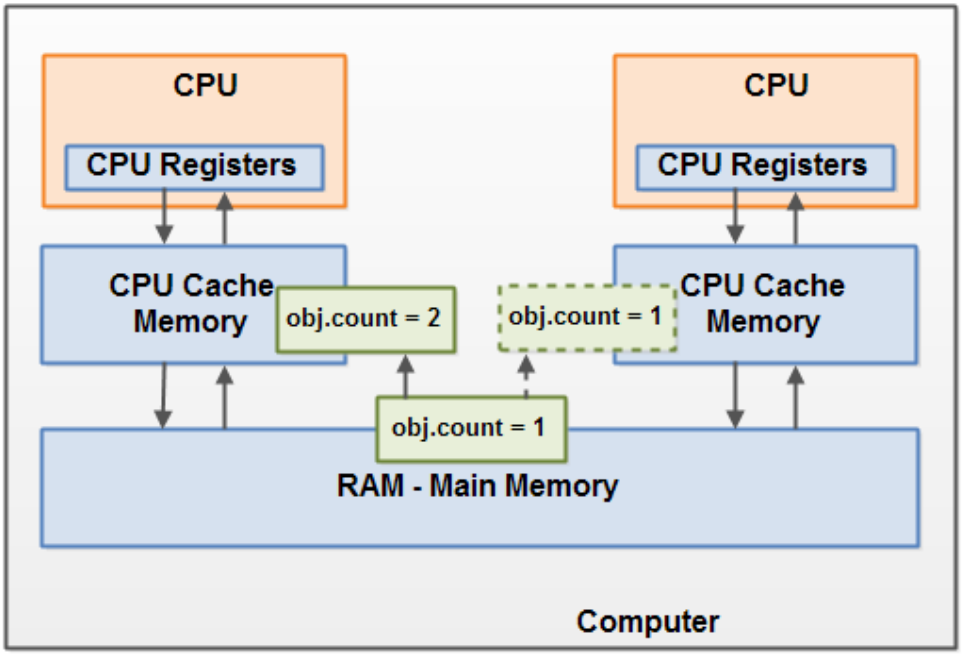
Řešení Javy:

Ad a) volatile - Tím se zajistí, že taková proměnná se čte rovnou z hlavní paměti, kam se také vždy zapíše při update.

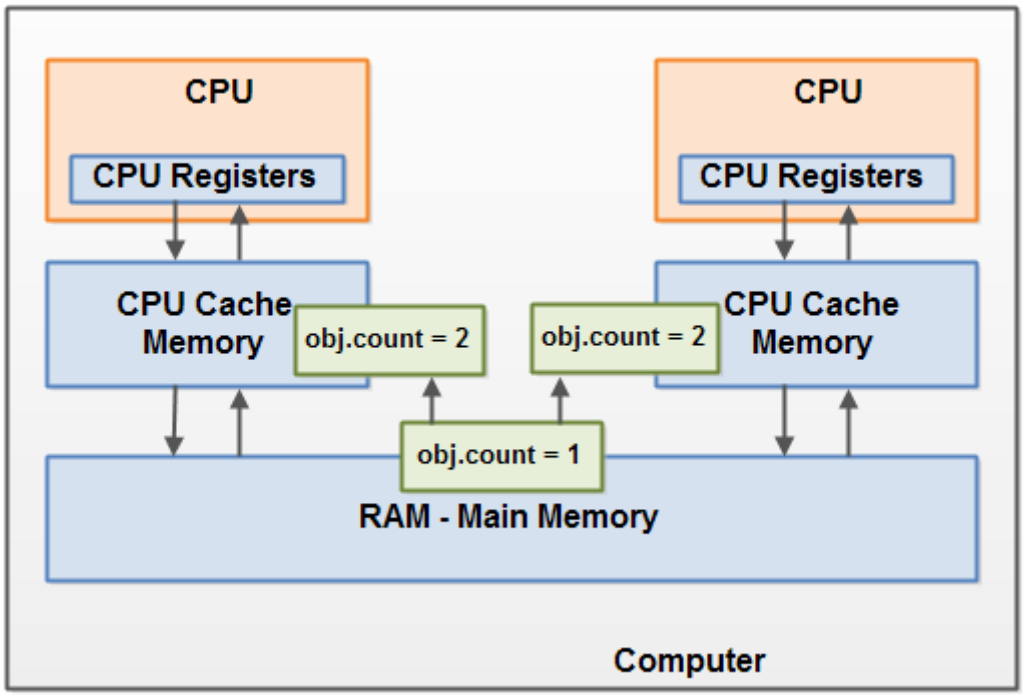
Ad b) synchronized = kritické sekce

- zaručují, že jen jedno vlákno může pracovat v kritické sekci programu,
- zaručují, že všechny proměnné, k nimž se přistupuje uvnitř bloku, budou přečteny do cache z RAM před zpracováním,
- zaručují, že když vlákno opouští synchr. blok, zapíší se všechny proměnné zpět do RAM, bez ohledu na deklaraci volatile nebo ne(volatile).

Př. 2 CPU, s běžícími vlákny, které sdílí objekt *obj* . Levá CPU natáhla objekt do své cache a změnila *obj.count* na 2. Pro vlákno z pravé CPU změna není viditelná, dokud nebude proveden zápis zpět do hlavní paměti..

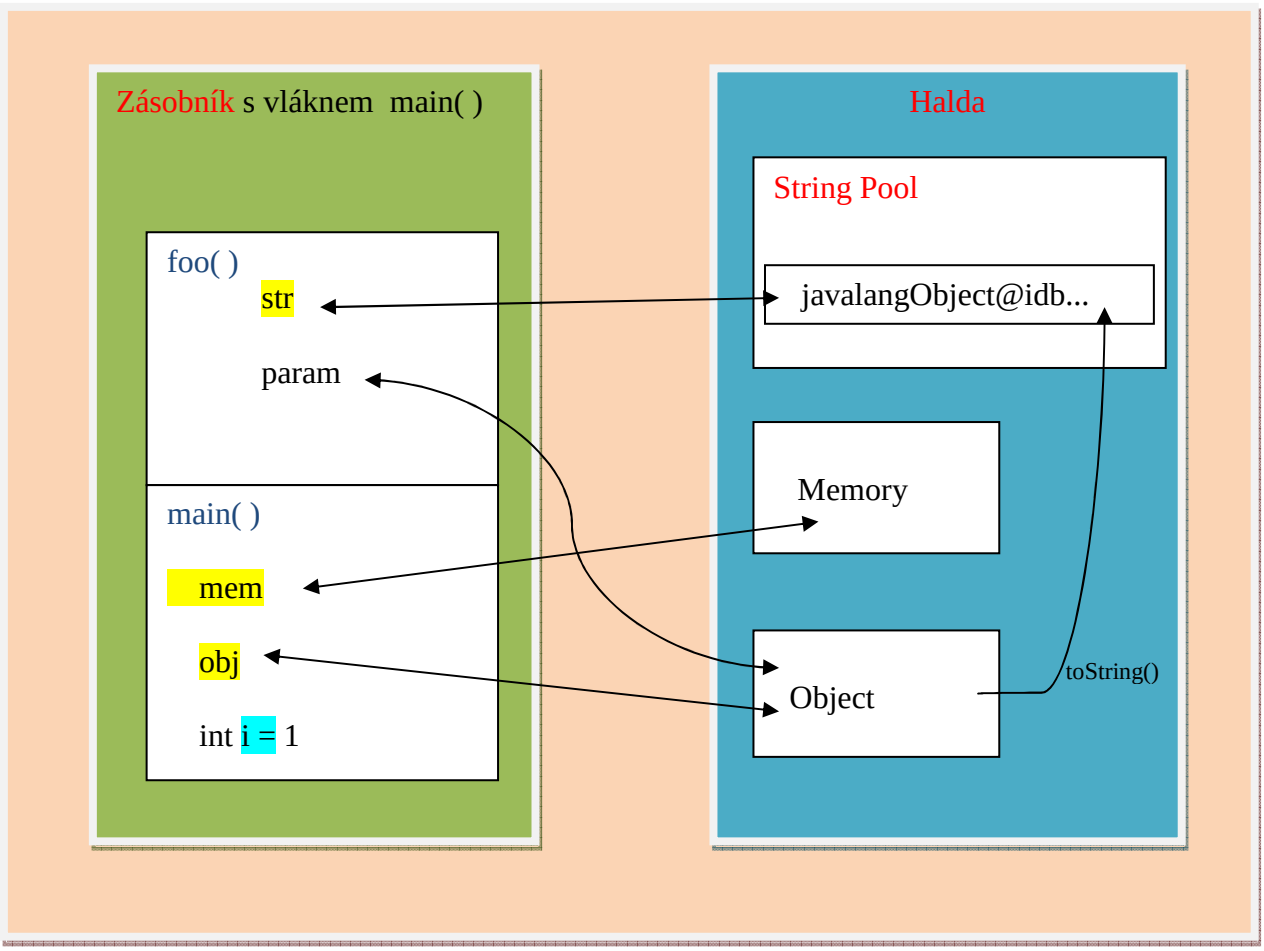


Př. vlákno Levé přečte proměnnou `count` sdíleného objektu do cache své CPU. Současně vlákno Pravé udělá totéž do své CPU cache. Pokud nyní jak Levé tak Pravé inkrementuje `count` Při sekvenčním provedení inkrementace by byla hodnota +2. Při souběžné exekuci obou na různých CPU bez synchronizace, bude po zápisu do hlavní paměti hodnota `count` +1.

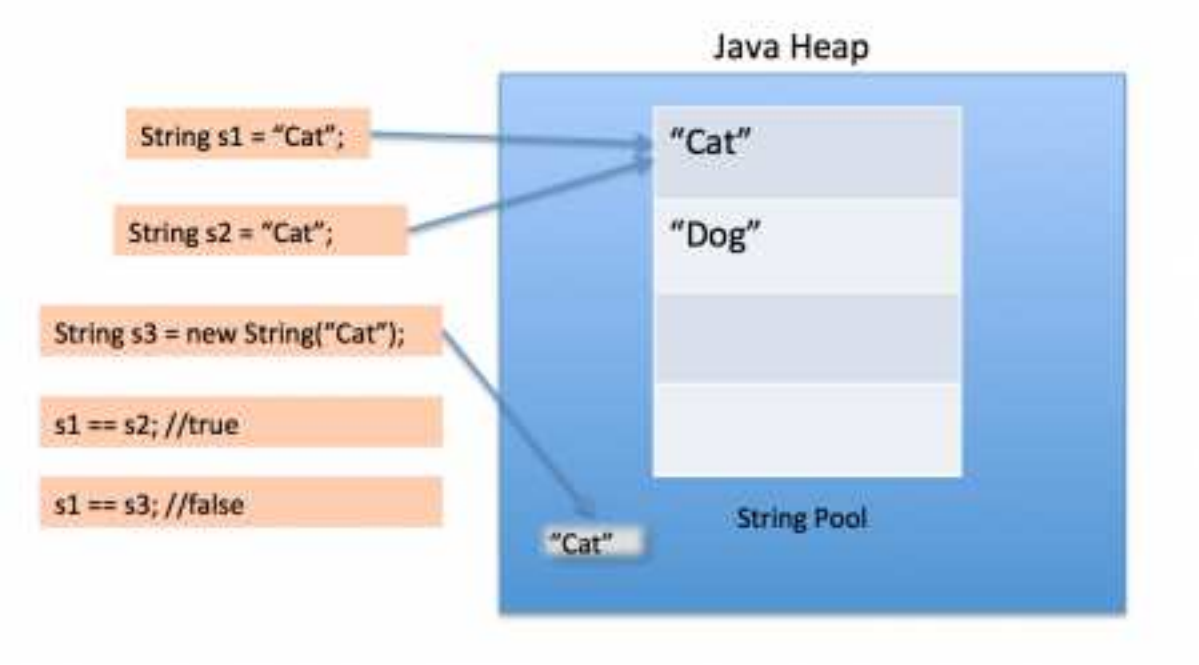


Př. Jak bude vypadat paměť v případě programu:

```
public class Memory {

      public static void main(String[] args) {// 1
            int i=1;                                  // 2
            Object obj = new Object();                // 3
            Memory mem = new Memory();                // 4
            mem.foo(obj);                             // 5
      }                                               // 9

      private void foo(Object param) {                // 6
            String str = param.toString();            // 7
            System.out.println(str);
      }                                               // 8
}
```



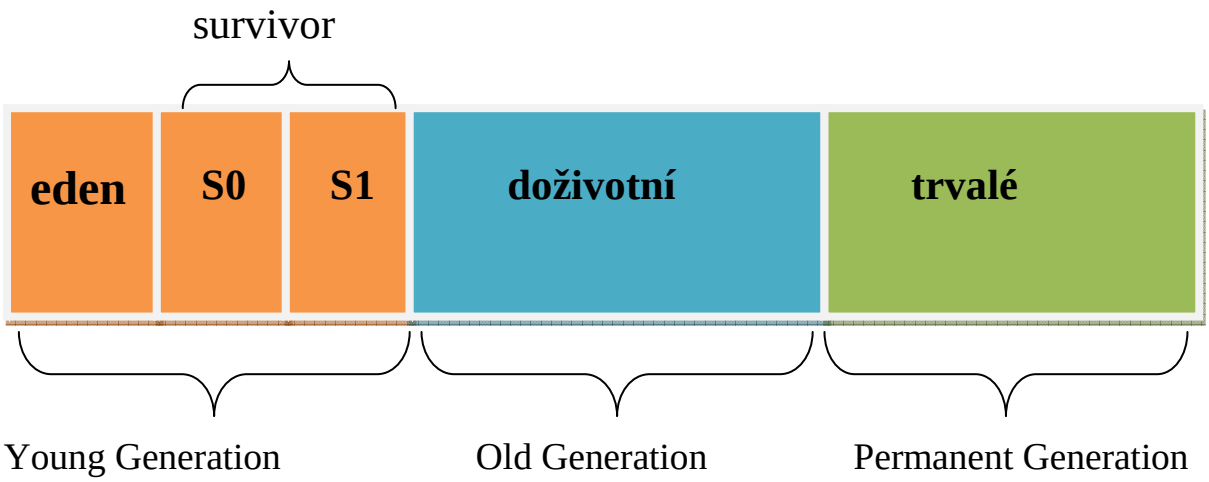
Způsob umisťování řetězců ukazuje obrázek



Shrnutí:

1. Haldu používají všechny části aplikace, zásobníky používají jen vlákna.
2. Při vytvoření objektu je tento uložen na haldě, zásobník obsahuje odkaz na něj, Kromě referenčních proměnných na objekty obsahuje zásobník lokální primitivní proměnné.
3. Objekty v haldě jsou globálně přístupné, zatímco zásobník vlákna není přístupný z jiných vláken.
4. Paměťový management zásobníků je LIFO, zatímco halda to má složitější s ohledem na globálnost přístupu. Detaily v Java Garbage Collection.
5. Zásobníky zanikají s vlákny, halda žije od startui do konce exekuce programu.
6. Můžeme definovat **-Xms** and **-Xmx** JVM opce pro startovací a maximální velikost haldy. Pomocí **-Xss** lze definovat velikost zásobníku.
7. java.lang.StackOverFlowError výjimka je vyhozena při plném stacku. Při plné haldě vyhazuje java.lang.OutOfMemoryError: Java Heap Space
8. Zásobník je mnohem rychlejší než halda. Paměť zásobníku je mnohem menší než paměť haldy.

Halda Javy



Části haldy:

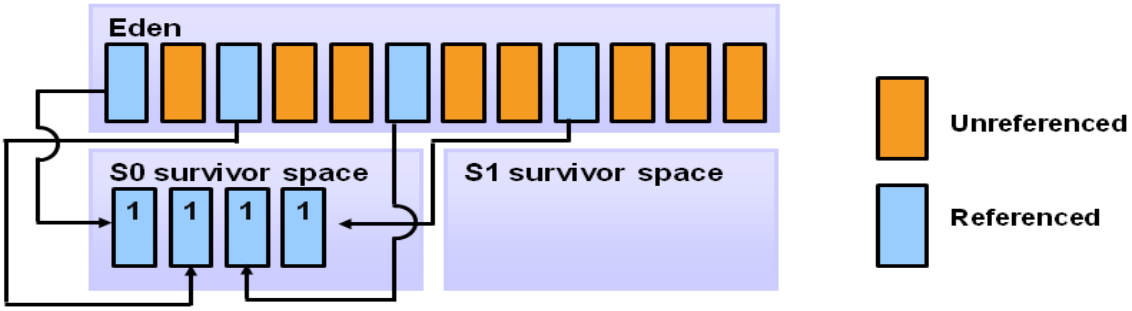
- Young Generation – pro nové objekty. Zaplnění spustí Minor Garbage Collection = Stop the Word Event.
- Old Generation – přesunuté dlouho přežívající objekty z Young Generation. Její čištění se zve Major Garbage Collection. = Stop the World Event.
- Permanent Generation – metadata pro JVM

Garbage Collection Process

Alokace - stárnutí - přesuny - čištění

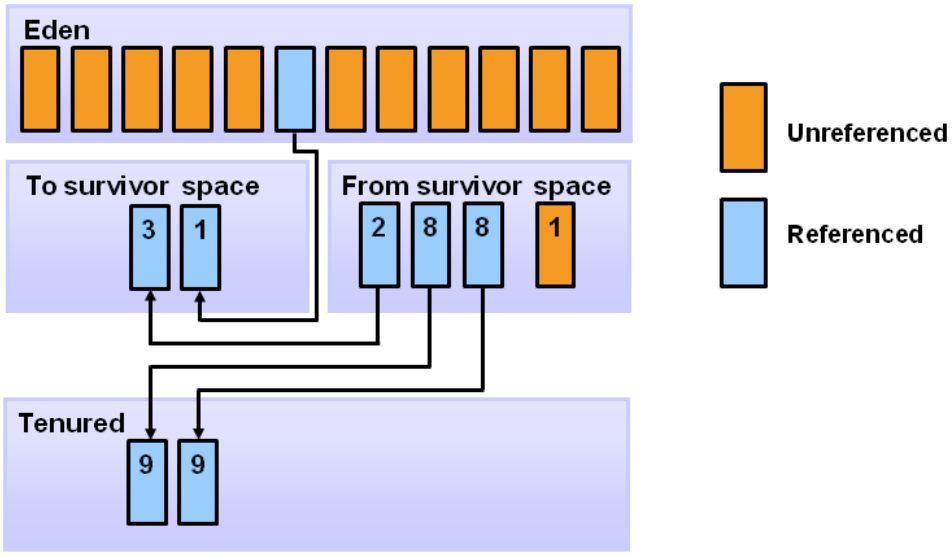
1. Každý nový objekt je umístěn v eden space. Oba survivor spaces jsou při startu prázdné.
2. Když se eden space zaplní, spustí se minor garbage collection.
3. Referencované objekty jsou přesunuty do S0 survivor space. Neodkazované jsou vypuštěny při čistění eden space.

Přesuny v young generation



4. Další minor GC probíhá obdobně, s rozdílem, že odkazované objekty jdou do S1 a objektům v S0 je inkrementován věk a jsou přesunuty do S1. Jakmile jsou všechny přeživší objekty v S1, tak S0 i eden jsou vyčištěny. Zůstávají nám v S1 objekty různého stáří.
5. V dalším minor GC se proces opakuje, S0 a S1 mají ale zaměněné role. Referencované objekty se přesouvají do S0. Přežívající objekty stárnou. Eden a S1 jsou vyčistěny.
6. Když věk objektů dosahne mezní hodnoty, jsou přendány z young generation do old generation (viz další obr.).

Propagace



7. Během minor GC exekucí objekty budou stárnout a propagují se do old generation space.
8. Eventuálně může dojít ke spuštění major GC na old generation. Ta vyčistí a převede do kompaktního stavu old generation space.