Cvičení 6 - Úvod do programování v BACI

- **paralelismus a synchronizační techniky** = základ OS, paralelního programování apod.

- **problém**
    - paralelnímu programování je obtížné porozumět bez praktické zkušenosti
    - časově závislé problémy se v reálných systémech projeví zřídka, proto je obtížné nalézt jejich zdroj

- proto **BACI** jako výukový prostředek pro získání zkušeností s paralelním programováním

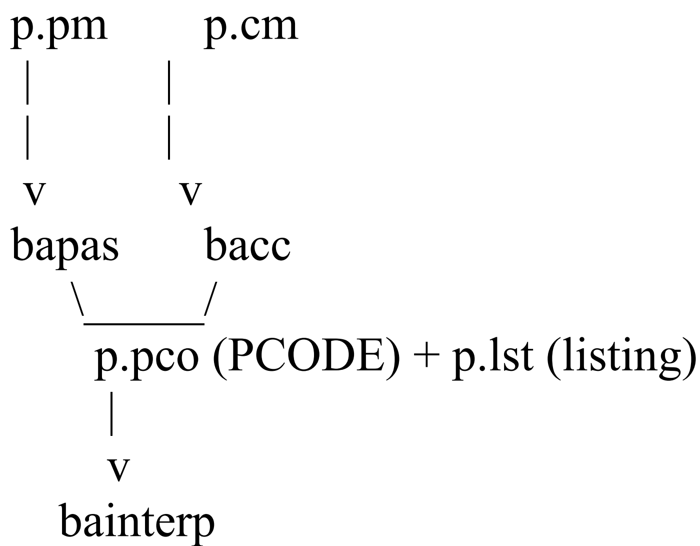
Co to je BACI?

- BACI = Ben Ari Concurrent Interpreter
- existuje pro Linux, MS DOS (může běžet i na příkazové řádce ve Windows)

- volně šířené, můžete si nainstalovat doma

- BACI obsahuje:
    - bacc a bapas - omezenou variantu překladačů C++ a Pascalu do PCODE
    - bainterp - interpret výsledného kódu, obsahuje možnosti ladění

Základní použití:



- další programy:
  - bagui - grafické rozhraní k interpretu (pouze pod systémy UNIX/Linux)
  - badis - disassembler, dekompiluje kód do čitelné podoby
  - baar a bald - program pro vytváření knihoven a linker.

..

BACI syntaxe pro C--

Poznámka:

BACI podporuje x = BACI umí přeložit a interpretovat konstrukci x
BACI nepodporuje x = BACI nerozumí x

1. **Komentáře** mohou být uzavřené v /* */ nebo je možné použít jednořádkové komentáře // jako v C++

2**. Funkce main()** musí mít jednu z následujících podob:

```
int main()
void main()
main()
```

Funkce main() musí být poslední fcí ve zdrojovém textu.

3. BACI nepodporuje jiné soubory než **standardní vstup a standardní výstup**: cin, cout a endl se chovají jako v C++, např.

```
cout << "hodnota i=" << i << endl;
```

4. Jediné **základní typy** v C-- jsou int a char. Oproti standardnímu C existují navíc typy týkající se *paralelismu*; tyto popíšeme později.

Všechny **proměnné** musejí být **deklarovány** na **začátku bloku**, kde se vyskytují *(tj. nelze definovat indexovou proměnnou v hlavičce smyčky for jako v C++)*

5. BACI podporuje typ string (řetězec). Při deklaraci řetězce musí být specifikována **délka řetězce**. Například následující deklarace zavádí řetězec o délce 20 znaků:

```
string[20] jméno_řetězce;
```

Délka řetězce je počet znaků řetězce (místo pro ukončující znak přidává překladač BACI). Délka musí být literál nebo konstanta.

V BACI se řetězce ukládají podobně jako v C do pole znaků a ukončují se znakem '\0'. **Kontrola přetečení** řetězců se **neprovádí**.

V deklaraci procedury nebo funkce může být klíčové slovo string použito pro deklaraci parametru typu string, například:

```
void foo(string jméno);
```

Parametr se předává **odkazem**, tj. řetězec musel být deklarován jako string[n] pro nějakou kladnou hodnotu n. Není možné volat fci foo("řetězec").

Pro práci s řetězci jsou k dispozici následující vestavěné funkce:

BACI	C	význam
stringCopy	strcpy	kopíruje řetězce
stringConcat	strcat	připojí řetězec na konec existujícího
stringCompare	strcmp	porovná abec. pořadí dvou řetězců
stringLength	strlen	vrací počet znaků řetězce
sscanf	sscanf	načte formátovaný řetězec
sprintf	sprintf	vypíše formátovaný řetězec

6. BACI podporuje **pole** libovolného platného typu, deklarace odpovídá standardnímu C:

```
platný_typ jméno_pole[index1][index2]...[indexN];
```

7. V C-- je podporováno klíčové slovo typedef; pokud chceme aby *"mujsem"* bylo synonymum s *"int"*, zapíšeme to jako

```
typedef int mujsem;
```

8. V C-- jsou podporovány **konstanty** jednoduchých typů:

```
const int m=5;
```

9. V deklaraci proměnných typu int a char je možné používat inicializátory:

```
const int m = 5;
int  j = m; // počáteční hodnota bude 5 atd.
int  k = 3;
char c = 'a';
```

10. Asi největší omezení: V BACI **není struct ani union.**

11. **Parametry** lze předávat **hodnotou i odkazem** jako v C++:

```
int afunc( int   a,     /* parametr předávaný hodnotou  */
           int&  b  )   /* parametr předávaný odkazem */
```

12. **Řídící konstrukce** jsou stejné jako v C/C++: if-else, switch/case, for, while, do-while, break a continue.

13. Do programu můžete **vkládat** zdrojové texty pomocí syntaxe:

```
#include <jméno_souboru>
#include "jméno_souboru"
```

Oba výše uvedené typy #include mají v BACI **stejnou sémantiku.**

Použití překladače a interpretu

Programování probíhá ve třech krocích:

1. Vytvoříme zdrojový text, např.:

```
main()
{
    int i;

    for(i=0; i<100; i++)
        cout << "a";
    cout << endl;
}
```

Soubor uložíme např. jako "p1.cm".

2. Soubor přeložíme:

```
$ bacc p1.cm
Pcode and tables are stored in p1.pco
Compilation listing is stored in p1.lst
$ _
```

Překlad vytvoří dva soubory - p1.pco (PCODE) a p1.lst (protokol o překladu).

3. Program **spustíme** zadáním jména programu (bez přípony .pco):

```
$ bainterp p1
```

Executing PCODE ...

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaa...aaaaaaaaaaaaaaaaaaaaaa
$ _

Syntaktické chyby

Pokud uděláme syntaktickou chybu, např. vynecháme přiřazovanou hodnotu:

```
main()
{
   int i;

   for(i=; i<100; i++)
      cout << "a";
   cout << endl;
}
```

Při překladu bude chyba ohlášena:

```
$ bacc p1
Error near ';', line 6 of file p1.cm:
**   parse error
Error near ')', line 6 of file p1.cm:
**   parse error
Because of 2 errors, the PCODE file will not execute
Pcode and tables are stored in p1.pco
Compilation listing is stored in p1.lst
$ _
```

Pokud je chyb více, je výhodné se podívat do protokolu o překladu (soubor s příponou .lst) a opravit podle něj všechny chyby, na které překladač upozornil.

```
BACI System: C-- to PCODE Compiler, 08:42  14 Jan 2003
Source file: p1.cm  Tue Jan 14 08:50:50 2003
 line  pc
   1    0
   2    0  main()
   3    1  {
   4    1      int i;
   5    1
   6    1      for(i=; i<100; i++)
Error near ';', line 6 of file p1.cm:
**   parse error
Error near ')', line 6 of file p1.cm:
**   parse error
   7   10          cout << "a";
   8   10      cout << endl;
   9   11  }
Because of 2 errors, the PCODE file will not execute
```

Sloupeček "line" říká číslo řádku ve zdrojovém textu, sloupeček "pc" je adresa, na které začíná vygenerovaná instrukce.

Při **pokusu spustit chybně přeložený program** vypíše obvykle bainterp ladicí informace:

```
$ bainterp p1
*** Halt at  2199 in process    0 because of invalid PCODE
... dalších 259 řádek chybového výpisu
```

Konstrukce pro paralelní programování

Konstrukce cobegin / coend

....

* v systému BACI se používají termíny **"vlákno" a "proces" jako synonyma**

* pro vytváření vláken se používá konstrukce typu **cobegin / coend**

```
  cobegin {
      p1(...); p2(...); ... ; pN(...);
  }
```

* hlavní program pokračuje **až po dokončení** všech vláken v bloku **cobegin**; vykonávání hlavního programu pokračuje za coend

* **instrukce** patřící vláknům jsou interpretem BACI náhodně prokládány, aby se program projevoval *nedeterministicky*

* **rozdíly** od *"abstraktních"* primitiv cobegin / coend:
 - v BACI může být cobegin použito pouze ve fci main()
 - konstrukce cobegin / coend není možné vnořovat
 - **"proces"** (resp. vlákno) musí v BACI být typu void

Poznámka:

Náhodné prokládání je důležitá vlastnost pro výukové účely. Při pseudoparalelním běhu je v reálně používaných systémech procesor každému procesu přidělen na delší dobu (protože přepínání má režii, tedy z důvodů efektivity). Časový souběh se pak projevuje podstatně řídčeji a proto se obtížněji odhaluje.

[]

Příklad 3 (demonstrace prokládání běhu dvou procedur)

* procedura "a" vypíše 100x znak 'a'
* procedura "b" vypíše 100x znak 'b'
* obě procedury spustíme paralelně pomocí cobegin / coend

// **p3.cm**

```
void a()
{
   int i;

   for(i=0; i<100; i++)
      cout << "a";
}

void b()
{
   int i;

   for(i=0; i<100; i++)
      cout << "b";
}
```

```
main()
{
    cobegin { a(); b(); }
    cout << endl;
}
```

[] --

Příklad 4 (demonstrace časového souběhu, domácí úkol)

* zavedeme si proměnnou x
* jedno vlákno proměnnou x zvětší 50x, druhé také

// **p4.cm**

```
int x=0;

void a()
{
    int i;

    for(i=0; i<50; i++)
        x++;
}

main()
{
    cobegin { a(); a(); }
    cout << "x=" << x << endl;
}
```

Domácí úkol:

Předpokládejte, že obě vlákna mohou běžet relativně vůči sobě libovolně rychle, k přepnutí mezi nimi může dojít libovolném okamžiku a že pro zvětšení hodnoty proměnné musí být tato hodnota nejprve zavedena do registru samostatnou strojovou instrukcí.

Určete spodní a horní mez pro koncovou hodnotu sdílené proměnné x, kterou může tento paralelní program vypsat.
(Nápověda: může to být i méně než 50.)

[]

Vestavěné funkce pro práci s řetězci:

* void stringCopy(string t, string z);

zkopíruje řetězec z do t, přetečení se nekontroluje (to platí i pro všechny ostatní procedury).

Například:

```
string t[20];
stringCopy(t, ""); // inicializuje na prázdný řetězec
```

* void stringConcat(string t, string z);

Připojí řetězec z na konec řetězce t.

* int stringCompare(string x, string y);

Porovnává abecední ("ASCIIbetické") pořadí řetězců x a y:

* pokud x má být zařazeno za y (x>y), vrací hodnotu větší než 0;
* pokud jsou oba řetězce totožné (x=y), vrací 0;
* pokud x má být zařazeno před y (x<y), vrací hodnotu menší než 0.

* int stringLength(string x);

Vrací délku řetězce x.

* int sscanf(string x, rawstring fmt, ...);

Postupně čte řetězec x a podle specifikace formátu fmt přiřazuje hodnotu proměnným uvedeným za fmt.

Funkce rozumí následujícím specifikacím formátu:

%d desítkové číslo
%x hexadecimální číslo
%s řetězec neobsahující mezeru
%q řetězec v uvozovkách; první nemezerový znak v řetězci musí být uvozovka (")

Například:

```
stringCopy(t,"123");      // mějme v t řetězec "123"
i = sscanf(t, "%d", j);   // v i bude počet zprac. položek (1)
                          // v j bude číslo 123
```

* void sprintf(string x, rawstring fmt,...);

Vytváří řetězec x podle specifikace formátu fmt a za ním následujících proměnných. Řetězec musí být dostatečně dlouhý, aby se do něj výsledek vešel.

Funkce rozumí následujícím specifikacím formátu:

%d desítkové číslo
%o oktalové číslo
%x hexadecimální číslo (cifry 10 až 15 se vypíšou jako a až f)
%X hexadecimální číslo (cifry 10 až 15 se vypíšou jako A až F)
%c znak
%s řetězec

Tyto specifikace formátu lze zadávat ve stejné obecnosti jako v ANSI C, tj. včetně specifikace šířky pole apod.

Například:

```
j:=333;
sprintf(t, "x=%d, tj. %o oktalove a %X hexadecimalne.", j, j, j);
cout << t << endl; // vypíše: x=333, tj. 515 oktalove a 14D hexadecimalne.
```