6. Variadické šablony. Paralelní programování.

B2B99PPC – Praktické programování v C/C++

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Šablony
  - Specializace
  - Typové aliasy (C++11)
  - Praktické příklady
  - Dědičnost
  - Variadické šablony
- Část 2 – Další konstrukce C++
  - Souborový systém
  - Structured binding
  - Měření času
- Část 3 – Paralelní programování
  - Procesy
  - Vlákna – Threads
  - Koncepty vícevláknových aplikací
  - Synchronizace vláken
  - Vlákna v C++ (STL)

Část I

Šablony

Šablony – opakování

- parametrizace typem nebo hodnotou
- instancování šablony při použití (parametrizací)
- existuje tolik instancí, kolik různých sad parametrů

```
1  template <typename T>
2  void print (const T & what)
3  {
4    std::cout << what << std::endl ;
5  }
7  print (20);
```

I. Šablony

Specializace

Typové aliasy (C++11)

Praktické příklady

Dědičnost

Variadické šablony

Úplná specializace šablony třídy

- Speciální implementace pro konkrétní šablonové parametry
- uvozená deklarací `template<>` za názvem třídy m konkrétní parametry opět v `< >`

```
1  template<typenameT>
2  class X  { ... };
4  template<>
5  class X <int> {
6    // speciální implementace pro int
7  };
```

- kromě tříd se můžou úplně specializovat
  - funkce (spíš nepoužívat)
  - metody šablonových tříd
  - statické atributy šablonových třída
  - další viz `http://en.cppreference.com/w/cpp/language/template_specialization`

Specializace metody šablonové třídy

```
1  #include <iostream>
3  template <typenameT, typename U>
4  class X {
5  public:
6    void foo() { std::cout << "nespecializovana\n"; }
7  };
9  template <>
10 void X <int, double>::foo() { std::cout << "specializovana!\n"; }
12 int main() {
13   X<int,int> x;
14   X<int,double> y;
15   x.foo();
16   y.foo();
17 }
```

Částečná specializace šablony

- uvozená deklarací `template<nespecializovane parametry>`
- nespecializované parametry je pak možno použít v seznamu parametrů za názvem třídy

```
1  template <typenameT,typenameU> classX {/* ... */};   // 1
2  template <typenameT> classX <T, double> {/* ... */}; // 2
3  template <typenameT> classX <T, T> {/* ... */};      // 3
4  template <typenameT> classX<int, T> {/* ... */};     // 4

7  X<double,char> x1; // použije se verze 1
8  X<char,double> x2; // použije se verze 2
9  X<char,char> x3;   // použije se verze 3
10 X<int,char> x4;    // použije se verze 4
11 X<int,int> x5;     // CHYBA! není jasné, kterou verzi použít
```

I. Šablony

Specializace

Typové aliasy (C++11)

Praktické příklady

Dědičnost

Variadické šablony

Šablonové

```
template <typename Element, typename Allocator>
class BasicContainer {/* ... */};
//
template <typename Element>
class StdAllocator {/* ... */};
//
template <typename Element>
using Container = BasicContainer<Element, StdAllocator<Element>>;
//
int main() {
  Container<double> c;
  // totéž, co BasicContainer<double, StdAllocator<double>>
}
```

Šablonové

- Fungují jako `typedef`, jen s jinou syntaxí

```
using VectorInt = std::vector<int>;
//
using Number = int;
//
using Function = void (*) (int, int);
//
template <typename T>
class Container {
  using valueType1 = T;
  typedef T valueType2;
};
```

I. Šablony

Specializace

Typové aliasy (C++11)

Praktické příklady

Dědičnost

Variadické šablony

STL algoritmy a funkční objekty

- Algoritmy v STL knihovně berou funkce/funktory jako parametry šablon

```
template<typename Iterator, typename Function>
void modify(Iterator from, Iterator to, Function func) {
  for (auto it = from; it != to; ++it) {
    *it = func(*it);
  }
}
```

Netypové parametry šablon 1/2

- parametry šablon mohou být i (některé) hodnoty
  - celočíselných typů (včetně bool, char)
  - výčtových typů
  - reference, ukazatele
- instance parametrů musí být konstantní výrazy

```
1 template <typename T, std::size_t N>
2 class array {
3   arr T[N];
4   // ...
5 };
```

Netypové parametry šablon 1/2

- Výpočet faktoriálu za překladu

```
1  template <unsigned N>
2  struct Factorial {
3    const static unsigned value = N * Factorial<N-1>::value;
4  };
6  template <>
7  struct Factorial<0> {
8    const static unsigned value = 1;
9  };
11 int main() {
12   // hodnota se spočítá při překladu, ne za běhu!
13   return Factorial<5>::value;
14 }
```

Netypové parametry šablon 2/2

- Parametrem šablony může být opět šablona

```
1  template <typename Value, template <typename> class Container>
2  class X {
3    Container<Value> cont;
4    // ...
5  };
6  // použití
7  X<int, std::vector> x;
8  // x obsahuje atribut cont typu std::vector<int>
```

Instance šablon 1/2

Kdy se vytvoří konkrétní instance šablony?

- během překladu
- pokud se v kódu objeví konkrétní použití
  - prototyp nebo použití funkce
  - použití třídy
  - explicitní instanciace
- pro instanciaci je třeba vidět celou definici šablonové třídy/funkce
- důsledek
  - buď inteligentní linker
  - nebo šablonové třídy a funkce v hlavičkovém souboru
- realita: šablony musí být v hlavičkovém souboru

Instance šablon 2/2

- Instance se vytváří jen pro typy, pro které jsou skutečně využity

```
1  template <typename T>
2  class X {
3    T t;
4  public:
5    void f() { t.f(); }
6    void g() { t.g(); }
7  };
9  class A { public: void f() {} };
11 int main() {
12   X<A> x;
13   x.f(); // takhle je to OK
14   // x.g(); // po odkomentování se nezkompiluje
15 }
```

Nápověda pro překladač – typename

```
1  template <typename T>
2  class Container {
3  public:
4    typedef T value_type;
5    using ptr_type = T *;
6    using size_type = unsigned int;
7    size_type size();
8  };
10 template <typename T>
11 void do_something(Container<T> & cont) {
12   // nebude fungovat:
13   Container<T>::size_type size = cont.size();
14   // je třeba napsat:
15   typename Container<T>::size_type size = cont.size();
16 }
```

I. Šablony

Specializace

Typové aliasy (C++11)

Praktické příklady

Dědičnost

Variadické šablony

Dědění šablon tříd

- Lze dědit parametrizovanou šablonu třídy
- Výsledkem je opět šablona nebo specializace

```
1  class CharVector : public std::vector <char>
2  {
3    // ...
4  };
6  template < typename T>
7  class BetterVector : public std::vector <T>
8  {
9    // ...
10 };
```

Konstruktor

- Je třeba ale dodefinovat konstruktor

```
struct ChVector : public std::vector<char> {
  ChVector (std::initializer_list<T> li) : std::vector<char> (li) {}
};
// ...
ChVector cv{1, 2, 3, 4};
```

- Nebo přebrat všechny konstruktory

```
stuct ChVector : public std::vector<char> {
  using std::vector<char>::vector;
};
```

- Pozn.: konstruktor šablonový není, jen třída

Obecné přebírání konstruktorů

- Existuje také obecnější způsob přebrání konstruktorů

```
1  class B : public A
2  {
3  public :
4    using A::A;
5  };
```

- Takto přejímáme vždy všechny
  - nelze přejmout jen vybrané
- Toto není výsadou pouze šablon, ale jakékoliv dědičnosti

I. Šablony

Specializace

Typové aliasy (C++11)

Praktické příklady

Dědičnost

Variadické šablony

Variadické šablony

- Dovolují definovat proměnlivý počet šablonových parametrů
- Podobná funkcionalita jako `va_args` z C
- Specializace
  - přímá (forwarding) – v případě, že není třeba přistupovat k jednotlivým argumentům, lze poslat celý balíček parametrů dál
  - rekurzivní – v případě, že je třeba přistupovat k jednotlivým parametrem
- Šablony funkcí a tříd

```
1 template <typename ... T>
2 void DoSomething (T ... args)
3 {
4   std::cout << " Doing ..." << std::endl ;
5   Something (args ...);
6 }
```

Parameter pack

- Syntaxe s třema tečkami – reprezentuje jeden a více šablonových parametrů

```
template <typename ... Args>
void DoSomething (Args ... args) {
  SomethingElse (args ...);
}
```

lec06/01-variadic-arguments.cpp

- Fold expressions – dovoluje aplikovat binární operátor na parameter pack (C++17)

```
template <typename ... Args>
void Print (Args &&... args) {
  (std::cout << ... << args) << std::endl ;
}
```

lec06/02-fold-expression.cpp

Počet argumentů

```cpp
#include <iostream>

template <typename ... Args>
void foo(Args... args) {
  std::cout << sizeof... (Args) << "\n";
}

int main()
{
  foo();                // 0
  foo(1);               // 1
  foo("string", 3.14); // 2

  return 0;
}
```

lec06/03-argument-count.cpp

Rekurzivní specializace 1/2

- Suma šablonou

```
1  template <typename T1>
2  T1 TemplateSum (T1 fa)
3  {
4    return fa;
5  }
7  template <typename T1, typename ... T>
8  T1 TemplateSum (T1 fa, T ... args)
9  {
10   return fa + TemplateSum (args ...);
11 }
13 TemplateSum (1, 2, 3, 4, 5);
```

Rekurzivní specializace 1/2

- Nemusí to být vždy úplně nejlepší nápad

```
TemplateSum (1, 2, 3, 4, 5);

// generuje všechny specializace funkce
TemplateSum <int>
TemplateSum <int, int>
TemplateSum <int, int, int>
TemplateSum <int, int, int, int>
TemplateSum <int, int, int, int, int>
```

- Důsledek: roste velikost binárky, v runtime 5 funkčních volání
- Někdy lze zredukovat použitím klíčového slova `inline`

Perfect forwarding

```
1  class Student {
2  public:
3    Student (const std::string & name, int birthyear) { ... }
4  };
5  //--
6  template <typename T1, typename T2>
7  Student MakeStudent (T1 && a, T2 && b) {
8    return Student (std::forward <T1>(a), std::forward <T2>(b));
9  }
10 //--
11 template <typename ... Args>
12 Student MakeStudent_2 (Args && ... args) {
13   return Student (std::forward<Args> (args)...);
14 }
```

Kontejnery založené na variadických šablonách

- `std::tuple` – uspořádaná n-tice
- hodnoty se vyzvedávají pomocí `std::tie` – přiřazení vnitřku do sady `l-value`
- nebo šablonově přes `std::get<n>`, kde `n` je index prvku

```
1  std::tuple<int , std::string, double> ntice;
3  int         a = std::get<0>(ntice);
4  std::string b = std::get<1>(ntice);
5  double      c = std::get<2>(ntice);
7  int a;
8  std::string b;
9  double c;
11 std :: tie (a,b,c) = ntice ;
```

Část II

Další konstrukce C++

II. Další konstrukce C++

Souborový systém

Structured binding

Měření času

std::filesystem

- práce se souborovým systémem, jednotné rozhraní (`<filesystem>`, od C++17)
  - práce se soubory, s adresári a odkazy
  - práva, časy modifikace, vytvoření, ...
  - obsazení a kapacita oddílu
- `std::filesystem::path`
  - reprezentuje cestu v souborovém systému
  - nemusí nutne identifikovat existující soubor/složku/odkaz
  - absolutní/relativní
  - prevod na absolutní `std::filesystem::absolute`
  - prevod na relativní `std::filesystem::relative`
  - lze implicitne prevést na `std::string`

```
1 std::filesystem::path filepath ("input.txt");
2 std::cout << filepath << std::endl;
3 std::cout << std::filesystem::absolute (filepath) << std::endl ;
```

std::filesystem::path

- připojení komponenty pomocí `append()` nebo operátorů `/` a `/=`

```
1 std::filesystem::path filepath ("C:\\");
2 filepath.append ("slozka");
3 filepath /= "podslozka";
```

- další metody
  - `filename` – vrací název souboru
  - `parent_path` – odebere poslední komponentu (vrací rodičovský adresář)
  - `make_preferred` – převod oddělovaču komponent cesty na preferované pro daný OS
  - `remove_filename` – odstraní z cesty název souboru
  - `replace_filename` – nahradí název souboru jiným názvem
  - `stem` – vrací název souboru bez přípony
  - `extension` – vrací příponu souboru

std::filesystem

- systémové cesty
  - `std::filesystem::current_path()` – vrací nebo nastavuje pracovní adresář
  - `std::filesystem::temp_directory_path()` – vrací adresář pro dočasné soubory
- manipulace se souborovým systémem
  - `std::filesystem::exists()` – existence souboru/složky/odkazu
  - `std::filesystem::copy()` – kopie složky nebo souboru
  - `std::filesystem::rename()` – prejmenování nebo presun
  - `std::filesystem::remove()` – smazání souboru nebo složky
  - `std::filesystem::status()` – vrací vlastnosti souboru
- `std::filesystem::directory_entry` – třída reprezentující položku v adresáři
  - metoda `path()` vrací fyzickou cestu
- `std::filesystem::directory_iterator` – iterování přes položky adresáře
- `std::filesystem::filesystem_error()`
  - přístup k souboru který neexistuje
  - systémové soubory, nedostatečná práva

II. Další konstrukce C++

Souborový systém

Structured binding

Měření času

Strukturovaná vazba (structured binding)

- lze svázat vícenásobnou inicializaci s nějakým objektem (od C++17)
- napr. rozkopírovat pole do více proměnných nebo nahradit `std::tie` u `std::tuple`
- lze svázat hodnotou nebo referencí

```
1  std::array <int, 2> arr {5, 10};
2  auto [a, b] = arr;
3  std::tuple <int, double> tup {5, 12.5};
4  auto &[i, d] = tup;
```

- možno např. ověřovat, zda se vložení prvku do kontejneru povedlo

```
1  std::map <int, std::string> mp;
2  if (auto [itr, succ] = mp.insert ({42, "meaning"}); succ)
3    std::cout << "OK" << std::endl ;
4  else
5    std::cout << "FAIL" << std::endl ;
```

Strukturovaná vazba

- lze svázat hodnotou, `l-value` nebo `r-value` referencí

```
std::array <int, 2> arr {5, 10};
auto [a, b] = arr;
auto & [c, d] = arr;
auto && [e, f] = std::make_tuple (5, 6);
const auto & [g, h] = std::make_tuple (7, 8);
```

- lze svázat s atributy objektu

```
struct S {
  int x; double y;
  S (int _x, double _y) : x(_x), y(_y) {}
};
S f(22, 3.14));

const auto [x, y] = f;
```

lec06/04-structured-binding.cpp

II. Další konstrukce C++

Souborový systém

Structured binding

Měření času

std::chrono

- standardizace časových jednotek a úseků, práce s časovými hodnotami (`<chrono>`)
- `std::chrono::duration<T>` – časový úsek reprezentovaný zvoleným typem
  - nezávislý na jednotkách (sekundy, minuty, ...)
- `std::chrono::steady_clock`
  - neklesající casovac, nereprezentuje reálný cas
  - konstantní doba mezi tiky → vhodný pro měření časových úseků, např. doba výpočtu

```
1  #include <chrono>
3  auto start = std::chrono::steady_clock::now();
4  std::cout << "f(42) = " << fibonacci(42) << '\n';
5  auto end = std::chrono::steady_clock::now();
6  std::chrono::duration<double> elapsed_seconds = end-start;
7  std::cout << "elapsed time: " << elapsed_seconds.count() << "s\n";
```

lec06/05-chrono-fibonacci.cpp

std::chrono

- `std::chrono::time_point<T>` – hodnota zvoleného časovače
- `std::chrono::system_clock`
  - reprezentuje reálný čas
- `std::chrono::high_resolution_clock`
  - časovač s vysokým rozlišením, nemusí reprezentovat reálný čas
- `now()` – metoda pro získání hodnoty
- `std::chrono::duration_cast<T>` – převod na požadované jednotky
- jmenný prostor `std::chrono_literals`
  - definované literály pro časové úseky (ns, us, ms, s, min, h)

```
1  using namespace std::chrono_literals;
3  auto duration = 150ms;
4  std::this_thread::sleep_for (250ms);
```

Část III

Paralelní programování

Paralelismus

- Opravdový paralelismus – několik HW systémů
- SW paralelismus (pseudo-parallelismus) – program s patřičnými knihovnami umožňuje spouštět paralelní úlohy na jednom nebo více procesorech

Proč psát paralelní programy?

- Větší výpočetní výkon
  - s více procesory je možné řešit složitější úlohy rychleji
- Efektivní využití výpočetního výkonu
  - v okamžiku, kdy program čeká na data – např. v případě, že program čeká na vstup od uživatele
- Současné řešení více požadavků
  - např. řešení požadavků klientů v architektuře klient-server

III. Paralelní programování

Procesy

Vlákna – Threads

Koncepty vícevláknových aplikací

Synchronizace vláken

Vlákna v C++ (STL)

Proces – program

- Proces je spuštěný program využívající přidělenou paměť
- Proces představuje záznam v tabulce spravované operačním systémem
  - OS rozvrhuje proces ke spuštění (scheduler) na dostupných procesorech
- Proces je běžně v jednom z následujících stavů:
  - Spuštěný – běží a využívá procesor
  - Blokovaný – čeká na periferii
  - Uspaný – čeká se na procesor a/nebo plánovač
- Proces má v rámci operačního systému přidělené svoje číslo PID

Více-procesorové systémy

- Umožňují naprogramovat pravý paralelismus
- Synchronizace běhu procesorů
  - Synchronizace aktivit
  - Synchronizace komunikace mezi procesy (běžícími na různých procesorech)

Architektury pro paralelismus

- Řízení jednotlivých instrukcí
  - **SIMD** – Single-Instruction, Multiple-Data
    - stejné instrukce jsou ve stejném okamžiku použity na rozdílná data
    - *procesory* jsou identické a běží synchronně
    - např. *vektorizace* jako MMX, SSE, 3Dnow! a AVX, atd.
  - **MIMD** – Multiple-Instruction, Multiple-Data
    - procesory běží nezávisle a asynchronně
- Přístup do paměti
  - Systémy se sdílenou pamětí – centrálně řízená sdílená paměť (multi-core CPUs)
  - Systémy s distribuovanou pamětí – každý procesor má svojí paměť (computational grids)

Role operačního systému

- OS poskytuje vrstvu HAL (Hardware Abstraction Layer)
  - zapouzdřuje HW zdroje
  - odděluje uživatele od detailů odlišujících jednotlivé HW realizace (true/pseudo paralelismus)
- OS je odpovědný za synchronizaci procesů
- OS poskytuje rozhraní (system calls) pro
  - Založení nebo zrušení procesu
  - Řízení procesů a procesorů
  - Plánování procesů na dostupné procesory
  - Funkce pro řízení přístupu do sdílené paměti
  - Mechanismus meziprocesorové komunikace IPC (Inter Process Communication)
  - Mechanismus pro synchronizaci procesů (čekání)

Programovací jazyky

- Bez přímé podpory paralelního programování
  - Paralelní programování je realizováno kompilátorem a operačním systémem
  - Paralelní konstrukce jsou značeny (direktiva `#pragma`, např. OpenMP)
  - Paralelní zpracování řeší OS pomocí systémových volání
- S přímou podporou paralelního programování
  - Obecně obsahuje podporu pro vytváření procesů
    - Běžící proces vytvoří svojí kopii
    - Oba procesy provádějí ty samé instrukce (kopie paměti)
    - Rodičovský proces a potomek jsou odlišení pomocí PID
    - V paměti vzniká nová oblast dedikovaná novému procesu – bitová kopie paměti
    - Lze vytvořit i proces jako zcela nový program – `fork`
  - Podpora řízení přístupu ke sdíleným zdrojům – např. semafory
    - přístup do tzv. kritické sekce – část programu, kde je potřeba zajistit exkluzivní přístup ke sdílenému prostředku
    - synchronizace dvou procesů
    - operace nad semafory musí být atomické

Komunikace mezi procesy – IPC (inter process comm.)

Zprávy

- Zasílání zpráv prostřednictvím tzv. front (message queues)
- Fronty jsou entity spravované OS

Sdílení paměti

- Část paměti se štítkem (label) dostupná z různých procesů
- Jedná se o část paměti, která patří OS!
- OS sbírá info o využití paměti a řídí přidělování práv

III. Paralelní programování

Procesy

Vlákna – Threads

Koncepty vícevláknových aplikací

Synchronizace vláken

Vlákna v C++ (STL)

Vlákna

- Vlákno je soubor instrukcí spouštěných nezávisle na hlavním procesu
  - Malý program zaměřený na specifickou část většího úkolu
- Vlákno běží uvnitř procesu
  - Sdílí tu samou paměť a paměťový prostor
- Vlákno má vlastní prostředí
  - Vlastní oblast pro proměnné
  - Vlastní identifikátor a prostor pro synchronizační primitiva
  - Vlastní Program counter (PC)/Instruction pointer (IP)
  - Vlastní oblast paměti pro lokální proměnné – Stack
- Vlákno může běžet v uživatelském prostoru nebo OS
  - Vlákna v uživatelském prostoru nepotřebují podporu OS, ale nemohou běžet současně
  - Vlákna běžící v OS mohou být plánována v rámci soutěže se všemi ostatními vlákny v systému, mohou běžet současně, ale vytváření vláken stojí čas

Kdy využívat vlákna

- Úloha obsahuje několik nezávislých částí
- Část úlohy může být blokována po určitý čas
- Úloha obsahuje výpočetně náročnou část (kde je potřeba stále interagovat s uživatelem)
- Aplikace musí rychle reagovat na asynchronní události
- Úloha obsahuje části s nižší a vyšší prioritou než zbytek aplikace
- Výpočetně náročná část může být urychlena paralelním zpracováním dat při využití více výpočetních jader

Typické aplikace využívající vláken

- Servery: obsluhují více uživatelů současně, využívají sdílené zdroje (databáze) a provádějí mnoho I/O operací
- Výpočetní aplikace: úspora času při využití více procesorů
- Aplikace reálného času: využití specifik plánovače – vícevláknová aplikace bude efektivnější než komplexní asynchronní program

Použití vláken

Efektivní využití výpočetních zdrojů

- Čekání na periferie → vlákno je blokované a řízení je předáno jiným vláknům
- Na systémech s více procesory/jádry je možné použít paralelní algoritmy

Řešení asynchronních situací

- Např. blokované vstupně/výstupní operace
- Procesor může řešit něco jiného (např. jedno vlákno řeší I/O operace z kamery a další vypočítává změny v obraze)

Vstupně výstupní operace

- Komunikace s periferiemi obsahuje hodně čekání – uživatelský vstup

Interakce s GUI

- Požadujeme okamžitou reakci na vstup uživatele – jinak se zdá, že systém zamrzl
- Uživatel v každém okamžiku generuje spoustu událostí, které ovlivňují aplikaci
- Výpočetně náročné operace by neměly ovlivnit interaktivnost aplikace – scrolování fotek

Vlákno a proces

Proces

- Výpočetní tok
- Vlastní paměťový prostor
- Součást OS
- IPC pomocí volání OS služeb
- CPU přidělovaný OS
- Vytvořit proces trvá

Vlákno (v procesu)

- Výpočetní tok
- Běží ve stejném paměťovém prostoru jako proces
- Uživatelská součást nebo součást OS
- Synchronizace pomocí exkluzivního přístupu k proměnným
- CPU je přidělován v rámci času dedikovaného mateřskému procesu
- Je rychlejší vytvořit vlákno než proces

III. Paralelní programování

Procesy

Vlákna – Threads

Koncepty vícevláknových aplikací

Synchronizace vláken

Vlákna v C++ (STL)

Boss/Worker

- Hlavní vlákno přijímá požadavky z vnějšku – zpracovává požadavky v cyklu
  - Přijme požadavek
  - Vytvoří/vybere pracovní vlákno, kterému přiřadí zpracování požadavku
  - Čeká na další požadavek
- Výsledek operace/požadavku je řízen
  - Buď přímo pracovním vláknem
  - Nebo hlavním vláknem, přičemž se použije nějaký synchronizační mechanismus

```
switch(getRequest()) {
  case taskX :
    create_thread(taskX);
    break;
  case taskY:
    create_thread(taskY);
    break;
}
```

```
taskX() {
  // solve the task
}
taskY() {
  // solve the task
}
```

Peer

- Neobsahuje řídicí vlákno
- První vlákno (procesu) vytvoří další vlákna a následně:
  - Se přepne do módu pracovního vlákna
  - Uspí samo sebe a čeká na ostatní vlákna
- Každé vlákno je odpovědné za svůj vstup a výstup

```
// 1st thread
create_thread(task1);
create_thread(task2);
.
.
start all threads;
wait to all threads;
```

```
task1() {
  wait to be exectued
  solve the task
}
task2() {
  wait to be exectued
  solve the task
}
```

Pipeline

- Dlouhý vstupní tok dat (stream) obsahují sekvence pro zpracování
  - Každý vstup musí být zpracován všemi částmi z pipeline
- V jednom časovém okamžiku jsou rozdílná vstupní data zpracována nezávislou částí

```
create_thread(stage1);
...
create_thread(stageN);
wait // for all pipeline
stage1() {
  while(input) {
    get next program input;
    process input;
    pass result to next the
    stage;
  }
}
```

```
stageN() {
  while(input) {
    get next input from
    thread;
    process input;
    pass result to output;
  }
}
```

Producent/Consumer

- Předávání dat mezi vlákny je realizováno pomocí bufferu (nebo ukazatelů)
- **Producer** – vlákno předávající data dalšímu vláknu
- **Consumer** – vlákno přijímající data z jiného vlákna
- Přístup do bufferu musí být synchronizovaný (exkluzivní přístup)

III. Paralelní programování

Procesy

Vlákna – Threads

Koncepty vícevláknových aplikací

Synchronizace vláken

Vlákna v C++ (STL)

Synchronizační mechanismus

- Vlákna používají obdobné mechanismy jako procesy
  - Protože vlákna sdílí paměť procesu, tak hlavní komunikační prostředek je paměť a globální proměnné
  - Kritický je přístup do stejné části paměti – je potřeba zajistit exkluzivní přístup do kritické sekce
- Základní synchronizační primitiva
  - Mutex/zámek – exkluzivní přístup do kritické sekce
  - Podmíněná proměnná (Conditional Variable) – sdílená proměnná umožňující synchronizaci vláken – spící vlákno může být probuzeno signálem z jiného vlákna

Základní synchronizační primitiva

- **Mutex** – sdílená proměnná, která je přístupná z ostatních vláken
- Poskytuje operace
  - Uzamknutí (lock) – mutex získalo vlákno, které mutex uzamklo
    - jestliže jiné vlákno požádá o stejný mutex (uzamčený), tak toto vlákno je systémem uspáno/blokováno a čeká na uvolnění mutexu
  - Odemčení (unlock) – lze provést na mutexu, který byl dříve zamčen
    - při odemčení se kontroluje, jestli na daný mutex nečeká jiné vlákno
    - pokud ano, tak se jedno z čekajících vláken vybere a dovolí se mu pokračovat v činnosti
- **Podmíněná proměnná** – umožňuje signalizovat z jednoho vlákna do druhého
- Poskytuje operace
  - Wait – proměnná je modifikovaná
  - Timed – čeká na signál z jiného vlákna
  - Signalizuje dalšímu vláknu změnu
  - Signalizuje všem čekajícím vláknům
    - Všechna vlákna jsou probuzena, ale protože přístup k podmíněné proměnné je chráněn mutexem, tak může mutex získat jen jedno vlákno

Příklad použití mutexu

- Zajištění exkluzivního přístupu k podmíněné proměnné z různých vláken

```
Mutex mtx; // shared variable for both threads
CondVariable cond; // shared condition variable
// Thread 1
Lock(mtx);
// Before code, wait for Thread 2
CondWait(cond, mtx); // wait for cond
... // Critical section
UnLock(mtx);
// Thread 2
Lock(mtx);
... // Critical section
CondSignal(cond, mtx); // signal on cond
UnLock(mtx);
```

Požadavky na funkce

- V případě paralelního zpracování, funkce jsou:
  - **Reentrant** – v jednom okamžiku je možné funkci spustit několikrát
  - **Thread-safe** – funkce může být volána různými vlákny ve stejný okamžik
- Jak to zajistit:
  - Reentrant funkce nepoužívá statická data a globální data
  - Thread-safe funkce striktně přistupuje ke globálním datům pomocí synchronizačních mechanismů

Problémy

- **Deadlock** – vlákno čeká na mutex, který je zamknutý jiným vláknem, přičemž jiné vlákno čeká na mutex zamčený prvním vláknem
- **Race condition** – přístup více vláken ke sdílené proměnné, kde minimálně jedno vlákno nepoužívá synchronizační prostředky

III. Paralelní programování

Procesy

Vlákna – Threads

Koncepty vícevláknových aplikací

Synchronizace vláken

Vlákna v C++ (STL)

std::thread

- Třída pro práci s vlákny (`<thread>`)
  - Spuštění funkce ve vláknu – `std::thread(function, args...)`
  - Čekání na vlákno – `std::thread::join()`
  - Odpojení vlákna – `std::thread::detach()`

```cpp
std::thread t(&thread_function); // startuje vlakno t
std::cout << "main thread\n";
t.join();  // hlavni vlakno ceka, az se t dokonci
/*
  vlakno t je mozne pustit jako samostatny proces (demon)
  t.detach() nelze kombinovat s join()
  vlakno nic nevypise, protoze hlavni program mezitim skonci
*/
return 0;
```

lec06/06-thread-join.cpp

std::mutex

- Třída pro práci s mutexy (<mutex>)
  - Zamknutí mutexu – std::mutex::lock()
  - Pokus o odemknutí – std::mutex::try_unlock()
  - Odemknutí mutexu – std::mutex::unlock()

```
1  std::mutex mu;
3  void shared_cout (std::string msg, int id) {
4    mu.lock();
5    std::cout << msg << ":" << id << std::endl;
6    mu.unlock();
7  }
9  std::thread t(&thread;_function);
10 for (int i = 100; i > 0; i--)
11   shared_cout("main thread", i);
12 t.join();
```

lec06/07-thread-mutex.cpp

Mutex a RAII

- Obecné ovládání mutexu (RAII princip)
  - `std::lock_guard (Lockable &m);`
  - `std:: lock_guard();`
  - `std::unique_lock();`

```
1  std::mutex mu;
3  void addToList(int max, int interval) {
4    std::lock_guard guard (mu);
5    for (int i = 0; i < max; i++) {
6      if((i % interval) == 0) myList.push_back(i);
7    }
8  }
10 std::thread t1(addToList, 100, 1);
11 std::thread t2(addToList, 100, 10);
```

lec06/08-thread-mutex-guard.cpp