Embedded programování. Paralelizace, multithreading, multitasking. Synchronizace a sdílené zdroje. Principy procedurálního programování. Vlastnosti jazyka C. Model kompilace, datové typy, ukazatele, standardní knihovna. (Programování v C)

# Embedded programování

Embedded (vestavěný) systém slouží k nějakému specifickému účelu v nějakém menším nebo větším celku. Příkladem pro větší systémy budiž systém termostatů pro kancelář nebo průmyslové budovy. Embedded systémy jsou implementovány převážně pomocí mikrokontrolérů s omezenými výpočetními a paměťovými specifikacemi. Embedded programování se tedy zabývá programováním softwaru pro tyto systémy s přihlédnutím k limitacím používaného hardwaru. Základem pro práci s embedded je porozumění vnitřní struktury a memory modelu konkrétního zařízení. Potřebné údaje o daném mikrokontroléru a zařízení získáme z jeho dokumentace

#### **Paralelizace**

Paralelizace označuje způsob výpočtu, kdy je prováděno více (často velmi mnoho) operací současně. Motivací je:

- Zvýšení výkonu výpočtu paralelní algoritmy mohou řešit výpočetní problémy rychleji
- Efektivní využití výkonu úkony jako čekání na vstup uživatele mohou blokovat vykonávání dalších částí programu
- Zpracování mnoha požadavků zároveň například v architektuře klient/server, kde od mnoha klientů (prohlížečů) najednou mohou přicházet požadavky na server (zdroj obsahu na webu)

Multiprocesorové (vícejádrové) systémy umožňují pravou paralelizaci. V tomto případě jsou potřeba tzv. resources (zdroje), které umožňují:

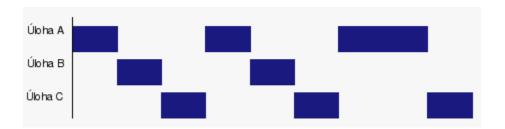
- Synchronizaci
- Komunikaci mezi procesy

### Multitasking

Proces je program, který běží a má vyhrazený prostor v paměti. Procesy mezi sebou tento prostor nesdílí. Proces se může nějakým způsobem pozastavi (např. čekání na periferii jako je tiskárna) a může tak blokovat chod systému. Multitasking je schopnost systému přepínat mezi procesy tak, aby to vypadalo, že běží současně. Procesor (CPU) vykonává procesy prokládaně, tj. vykonává sekvence instrukcí z různých procesů za sebou. Multitasking tedy funguje tak, že je přerušeno vykonávání jednoho procesu, uloží se jeho stav (mezivýsledky, obsah v paměti, v registrech procesoru), načte se stav jiného procesu a tomu se předá možnost vykonávání instrukcí v procesoru. Toto přidělování výpočetního výkonu procesoru dělá Scheduler (součást operačního systému).

Multitasking nevyžaduje paralelní vykonávání, tj. stačí jeden procesor (jedno jádro) k vykonávání všech procesů zdánlivě současně. Vícejádrové procesory dovolují větší výkon díky pravé paralelizaci, ale multitasking je zde využit úplně stejně (takže opět může běžet mnohem více úloh, než je jader procesoru).

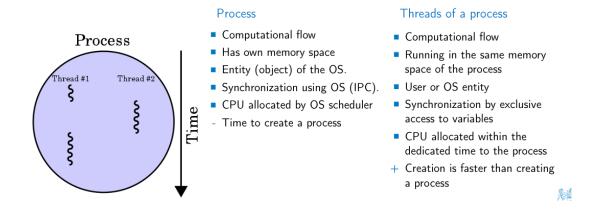
Vzniklo tak paradigma, že je pro jeden úkol víc procesů a ty spolu třeba různě spolupracují, vznikají a zanikají. Ale tvorba procesu je relativně drahá (rozuměj rezervovat místo v paměti pro proces je relativně pomalé). Proto existují vlákna.



## Multithreading

Vlákno je součást procesu, sekvence instrukcí, která sdílí paměť s ostatními vlákny uvnitř procesu. Typicky se soustředí na nějaký úkol v rámci celé aplikace, aby se program nezasekl. Výhoda vláken je, že se tvoří rychleji, než nové procesy, protože prostě sdílí paměť s ostatními vlákny v procesu a není jim nově přidělována. Jinak se jedná typ multitaskingu, akorát tady se specificky multitaskují vlákna a ne procesy. Rozdíl mezi procesy a vlákny je i ten, že **procesy** jsou synchronizovány a výkon na procesoru je jim přidělován **operačním systémem** přímo. Přehled v obrázku níže.

# Threads and Processes



## **Synchronizace**

# Pozn.: přístup k paměti znamená zápis nebo čtení.

Synchronizace **procesů** - v programech jsou části kódu jménem kritické sekce. V těchto sekcích má proces přistupovat ke sdílené paměti. Je třeba zabránit tomu, aby dva a více procesů přistupovali najednou (představte si, že dva lidé chtějí najednou nakrájet jednu bramboru - navíc třeba jeden napříč a druhý podélně - katastrofa!). Proto existují synchronizační techniky, které organizují povolení k přístupu k paměti. Když jeden proces přistupuje k paměti, ostatní musí čekat. Často musí taky být procesy vykonávány v přesném pořadí. Když se synchronizace neimplementuje nebo je špatně udělaná, mohou být hodnoty různých (i kritických) proměnných nepředvídatelné. To může vést k tzv race **condition/deadlock**, kdy se program zasekne (například dva procesy na sebe čekají navzájem). Analogicky pro vlákna.

**Semafory** jsou proměnné určené pro synchronizaci. Uvažujme příklad z wikipedie:

Consider a system that can only support ten users (S=10). Whenever a user logs in, P is called, decrementing the semaphore S by 1. Whenever a user logs out, V is called, incrementing S by 1 representing a login slot that has become available. When S is 0, any users wishing to log in must wait until S>0 and the login request is enqueued onto a FIFO queue.

# Sdílené zdroje (paměť)

Označená část paměti, která je přístupná více procesům. Operační systém spravuje **informaci** o užívání sdílené paměti, a povolení a práva přístupu k této paměti. V céčku jsou dostupné system calls (systémová volání, funkce), které slouží k obsluze sdílené paměti.

Např. shmget (získání indentifikátoru sdílené paměti), shmread, shmwrite.

## Principy procedurálního programování

Založené na volání procedur. Procedury obsahují seřazené kroky k vykonání. Funkce jsou procedurami. Přijmou vstup, vykonají nějaké kroky (např. spočtou nebo modifikují) a vrátí (hodnotu nebo nic).

Výhoda procedur je modularita, program lze rozčlenit do více funkcí, které jde opakovaně volat, recyklovat. To je úspora a vyvarování se chybě oproti opakovanému (duplicitnímu) kódu. Celkově to splňuje princip DRY (don't repeat yourself).

Další výhodou týkající se modularity je tzv enkapsulace, tj že **proměnné vytvořené uvnitř funkce** (včetně těch deklarovaných v argumentu - např int add(int a, int b)) **existují jen po dobu vykonávání funkce -** jak končí funkce, končí i proměnná (uvolní se paměť atd). Tyto proměnné navíc nesouvisí s proměnnými mimo funkci, takže když někde ve funkci main vytvořím proměnnou a, tak ta je jiná, než proměnná a třeba ve funkci int add(int a, int b).

Existence funkcí taky vede k tvorbě knihoven - sbírek funkcí, které často plní specifický účel. Například vykreslování obrázků nebo síťová komunikace. Knihovny jsou sdíleny mezi více uživateli (programátory), aby nemuseli znovu "vynalézat kolo".

# Vlastnosti jazyka C

Zejména systémový jazyk (pro psaní softwaru pro operační systémy, včetně psaní samotných operačních systémů), ale je použitelný obecně pro aplikace (general purpose language).

Je nízkoúrovňový, to znamená, že v něm je relativně málo abstrakcí (zjednodušení) oproti strojovému jazyku (jazyk assembly). Taky se někdy říká "blízko k hardwaru". Dá se v něm hodně přímo manipulovat s počítačovou pamětí, což je u jazyků vyšší úrovně většinou skryto a nedělá to přímo programátor.

Ukládání dat je v zásobníku a v haldě. Alokace dat v zásobníku je automatická, ale statická - třeba pole má fixní délku. To řeší dynamická paměť halda, kde můžeme délku dynamicky měnit, ale musíme se o to postarat sami (především o uvolnění paměti, když proměnnou přestaneme používat. Jinak se paměť sama neuvolní a bude zabraná až do konce běhu programu - tzv memory leak). Chyba je taky double free - když zavoláme free na již uvolněnou paměť.

Jazyk C má všechny očekávané kontrolní struktury jako if, else if, else. Pak iterativní struktury jako for a while. Je to procedurální jazyk.

Je kompilovaný. To znamená, že je se céčkový kód musí nejdříve zpracovat, aby se z něj stal spustitelný program. **Model kompilace** je:

Zdrojový kód > preprocessing: zde se vykonávají různé direktivy s #, třeba #include > kompilace: výstupem jsou objektové soubory > linkování: objektové soubory se spojí dohromady (třeba ještě už se zkompilovanými knihovnami) do spustitelného souboru.

- Preprocesor (dosazuje kusy kódu za includy, # makra, vyhazuje komentáře, spojuje řádky dohromady když je tam "/"). Pokud chceme spustit pouze fázi preprocessingu, přidáme přepínač
  E do kompilace.
- Kompilace (překlad) (kompiluje převede programátorem napsaný zdrojový kód do ".o" (binární) podoby. To jsou soubory, které již obsahují binární kód. Tento kód ale není spustitelný, protože nemá vyřešené závislosti na jiné části programu (například volání funkce, která je v jiném .c souboru). Je třeba znát deklarace, vkládá jen relativní adresy.
- Linkování (vkládají se absolutní hodnoty adres proměnné a funkce, výsledkem je spustitelný program). Při kompilování do objektových souborů kompilátor neví, zda volaná funkce existuje a kde je. Použití správných adres v paměti řeší linker.

## Datové typy:

Co se týče obecně vytváření a manipulace s proměnnými.

int a; → tohle je deklarace

a = 5;  $\rightarrow$  tohle je inicializace.

int b = 11; → tohle je deklarace a inicializace

## Typy - k čemu se používají a velikosti

Typy jsou buď znamínkové nebo neznamínkové (signed and unsigned). Rozsah každého typu záleží podle je signed nebo unsigned. Tj. obecně je rozsah pro signed  $-2^{bits/2}$  až  $2^{bits/2}-1$ , kde bits je velikost typu v bitech. Pro int je range  $-2^{32/2}$  až  $2^{32/2}-1$ , tj. -32 768 až 32 767. Pro unsigned typy je rozsah 0 až  $2^{bits}-1$ . Takže pro unsigned int je rozsah 0 až  $2^{32}-1$  (=65535). To -1 tam je proto, že do toho rozsahu se počítá i nula.

#### Základní:

- Char k reprezentaci symbolů, 8 bits (1 byte)
- o Int k reprezentaci celých čísel, 32 bits (4 bytes)
- Long samý jako Int, ale dvakrát větší, 64 bits (8 bytes)
- Float čísla s desetinnou čárkou, 32 bits
- Double samý jako float, jen dvakrát větší, 64 bits

### Složené:

- o pole (fixní kolekce věcí stejného typu),
- enum (výčtový typ, používá se často pro options pro funkci. Třeba když chce funkce vědět barvu, tak můžete udělat enum COLORS = {NONE, BLACK, BLUE, ...} a funkce ho bude přijímat v argumentu),
- struct (do struktury můžete ukládat více proměnných různých typů, včetně dalších struktur atd. K těm pak můžete přistupovat šipkovou syntaxí struktura->proměnná)

#### Ukazatele

Ukazatele jsou proměnné, které obsahují ne přímo hodnotu, ale adresu v paměti. Chceme-li například uložit adresu nějakého intu, řekněme proměnné *a*, napíšeme int\* p = &a;

Hvězdička za datovým typem značí, že se jedná o ukazatel, ampersand (&) je operátor, který nám dává adresu proměnné, před kterou je. V lidštině by se to řeklo takto: Chci ukazatel jménem *p*, který bude ukazovat na int a dám mu hodnotu, kterou je adresa proměnné *a*.

Proměnná a bude mít třeba hodnotu 69 a adresu 0x420. V tom případě bude mít pointer *p* hodnotu 0x420. Když máme pointer a chceme zjistit co se skrývá na adrese, kterou obsahuje,

použijeme hvězdičku před: \*p nám dá 69. Chceme-li tu hodnotu upravit tak použijeme opět to samé: \*p = 1337;

Je třeba rozlišovat význam hvězdičky v různých kontextech, abyste se vyhnuli zmatku! Hvězdička při deklaraci **mezi typem a názvem proměnné** (jedno kde) prostě říká, že ta proměnná je pointer. Tj například:

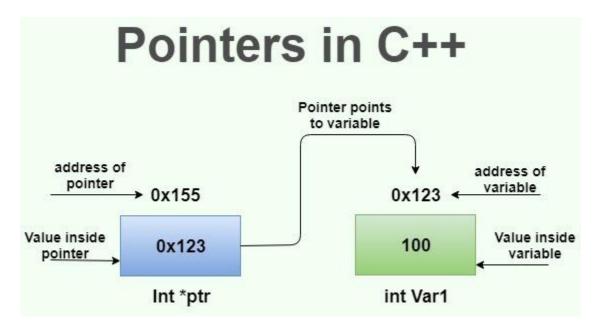
int\* p;

int \* p;

int \*p;

Znamenají to samé a to deklaraci pointeru.

Ukazatel jsou velice užitečný nástroj, ale je třeba s nimi náležitě pracovat. Když máme ukazatel na dynamicky alokovanou paměť, je třeba mít ošetřeno uvolnění paměti, a taky se vyvarovat věcem jako double freeing - uvolňování již uvolněné paměti (vyústí v runtime error - **runtime** označuje běh programu).



#### Standardní knihovna

Knihovna - Kolekce kódu poskytují různé funkcionality. Důvodů pro knihovny je několik. Nechceme psát vše od znova pokaždé, když píšeme nějaký program. Třeba když chci udělat aplikaci s grafickým rozhraním, nebudu psát všechny věci jako vykreslování okna, prvky jako tlačítka atd vymýšlet znovu, ale najdu si knihovnu která bude mít funkci pro vykreslení okna, kde si jen zadám rozměry, atd.

Standardní knihovna jazyka je sbírka funkcí pro základní operace i interakci se systémem a tak dále. Je většinou definována nějakým standardem/specifikací (to znamená, že nějaká komise vymyslí standard, třeba že v další verzi jazyka přibude nějaká funkce, a pak je na někom, aby ho implementoval).

Standardní knihovna C obsahuje definice typů, makra, funkce například pro manipulaci se stringy, matematické operace a tak dále.

Důležitá knihovna je stdlib.h, která definuje různé důležité typy a funkce, například včetně malloc, realloc, free, které umožňují práci s dynamickou pamětí. Další hlavní části knihovny jsou třeba stdio.h, což znamená standard input output, a složí třeba pro výpis do konzole nebo naopak vstup. Math.h má funkce jako sin, cos, log a další.

Věci jako funkce pro vlákna nejsou ve standardní knihovně, ale jsou definované v systému.