

B4B35OSY: Operační systémy

Lekce 3. Procesy a vlákna

Petr Štěpán
stepan@fel.cvut.cz



10. září, 2020

Outline

1 Proces

2 Vlákna

3 Od programu k procesu

Obsah

1 Proces

2 Vlákna

3 Od programu k procesu

Proces

- Výpočetní proces (job, task) – spuštěný program
- Proces je identifikovatelný jednoznačným číslem v každém okamžiku své existence
 - PID – Process IDentifier
- Co tvoří proces:
 - Obsahy registrů procesoru (čítač instrukcí, ukazatel zásobníku, příznaky FLAGS, uživatelské registry, FPU registry)
 - Otevřené soubory
 - Použitá paměť:
 - Zásobník – .stack
 - Data – .data
 - Program – .text
- V systémech podporujících vlákna bývá proces chápán jako obal či hostitel svých vláken

Proces – požadavky na OS

- Umožňovat procesům vytváření a spouštění dalších procesů
- Prokládat – „paralelizovat“ vykonávání jednotlivých procesů s cílem maximálního využití procesoru/ů
- Minimalizovat dobu odezvy procesu prokládáním běhů procesů
- Přidělovat procesům požadované systémové prostředky
 - Soubory, V/V zařízení, synchronizační prostředky
- Umožňovat vzájemnou komunikaci mezi procesy
- Poskytovat aplikačním procesům funkčně bohaté, bezpečné a konzistentní rozhraní k systémovým službám
 - Systémová volání – minulá přednáška

Vznik procesu

- Rodičovský proces vytváří procesy-potomky
 - pomocí služby OS. Potomci mohou vystupovat v roli rodičů a vytvářet další potomky, ...
 - vzniká tak strom procesů
- Sdílení zdrojů mezi rodiči a potomky:
 - rodič a potomek mohou sdílet všechny zdroje původně vlastněné rodičem (obvyklá situace v POSIXových systémech)
 - potomek může sdílet s rodičem podmnožinu zdrojů rodičem k tomu účelu vyčleněnou
 - potomek a rodič jsou plně samostatné procesy, nesdílí žádný zdroj
- Souběh mezi rodiči a potomky:
 - Možnost 1: rodič čeká na dokončení potomka
 - Možnost 2: rodič a potomek mohou běžet souběžně
- V POSIXových systémech je každý proces potomkem jiného procesu
 - Výjimka: proces č. 1 (init, systemd, ...) vytvořen při spuštění systému
 - Spustí řadu procesů a skriptů (rc), ty inicializují celý systém a vytvoří démony (procesy běžící na pozadí bez přístupu na terminál) service ve Win32
 - Init spustí také
 - textové terminály proces *getty*, který čeká na uživatele → login → uživatelův shell
 - grafické terminály *display manager* a *greeter* (grafický login)

POSIX vytvoření procesu – fork/exec

Rodič vytváří nový proces – potomka – voláním služby **fork**

- vznikne identická kopie rodičovského procesu až na:
 - návratovou hodnotu systémového volání
 - hodnotu PID, PPID – číslo rodičovského procesu
- návratová hodnota určuje, kdo je potomek a kdo rodič
 - 0 – jsem potomek
 - PID – jsem rodič a získávám PID potomka
- vytvoří se virtuální kopie programu – fyzicky je vykonáván program shodný s programem rodiče
- potomek může použít volání služby **exec** pro náhradu programu ve svém adresním prostoru jiným programem
 - Příklad – bash použije funkci fork pro vytvoření potomka, kterého pak nahradí zadaným programem.

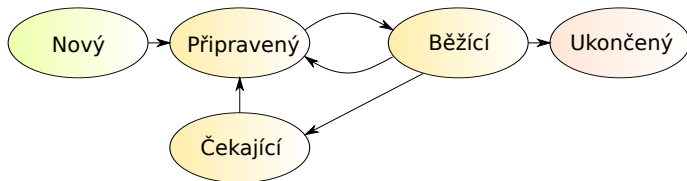
Ukončení procesu

- Proces provede poslední instrukci programu a žádá OS o ukončení voláním služby `exit(status)`
 - Stavová data procesu-potomka (`status`) se musí předat procesu-rodíči, který typicky čeká na potomka pomocí `wait()`
 - Zdroje (paměť, otevřené soubory) končícího procesu jádro samo uvolní
- Proces může skončit také:
 - přílišným nárokem na paměť (požadované množství paměti není a nebude k dispozici) – *OOM killer* v Linux
 - běžící kód vygeneruje výjimku CPU, kterou jádro neumí vyřešit:
 - aritmetickou chybou (dělení nulou, `arcsin(2)`, ...)
 - pokusem o narušení ochrany paměti („zabloudění“ programu)
 - pokusem o provedení nedovolené (privilegované) operace (zakázaný přístup k hardwarovému prostředku)
 - ...
 - žádostí rodičovského procesu (v POSIXu signál)
 - Může tak docházet ke kaskádnímu ukončování procesů
 - V POSIXu lze proces „odpojit“ od rodiče – démon
 - a v mnoha dalších chybových situacích

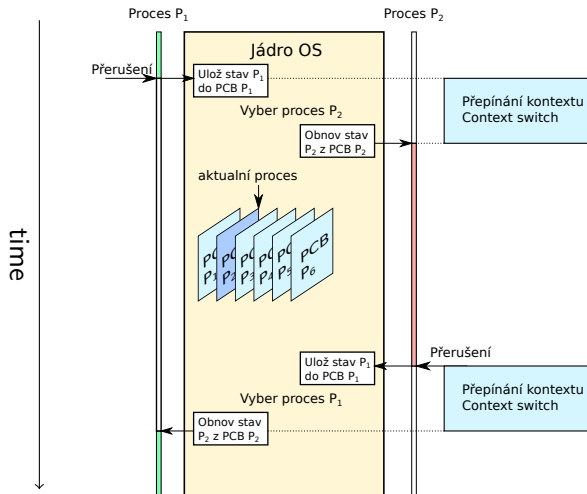
Stav procesu

Proces se za dobu své existence prochází více stavy a nachází se vždy v jednom z následujících stavů:

- **Nový (new)** – proces je právě vytvářen, ještě není připraven k běhu, ale již jsou připraveny některé části
- **Připravený (ready)** – proces čeká na přidělení procesoru
- **Běžící (running)** – instrukce procesu je právě vykonávány procesorem, tj. interpretovány některým procesorem
- **Čekající (waiting, blocked)** – proces čeká na událost
- **Ukončený (terminated)** – proces ukončil svoji činnost, avšak stále ještě vlastní některé systémové prostředky



Přepínání procesů



Přepínání procesů

- Přejed od procesu P_1 k P_2 zahrnuje tzv. přepnutí kontextu
- Přepnutí od jednoho procesu k jinému nastává výhradně v důsledku nějakého přerušení (či výjimky)
- Proces P_1 přejde do jádra operačního systému, který provede přepnutí kontextu \rightarrow spustí se proces P_2
 - Nejprve OS uschová stav původně běžícího procesu P_1 v $PCBP_1$
 - jádro OS rozhodne, který proces poběží dál – P_2
 - Obnoví se stav procesu P_2 z $PCB P_2$
- Přepnutí kontextu představuje režijní ztrátu
 - během přepínání systém nedělá nic užitečného, nepoběží žádný proces
 - časově nejnáročnější je správa paměti dotčených procesů
- Doba přepnutí závisí na hardwarové podpoře v procesoru
 - minimální hardwarová podpora je implementace přerušení:
 - uchování IP a FLAGS
 - naplnění IP a FLAGS ze zadaných hodnot
 - lepší podpora:
 - ukládání a obnova více/všech registrů procesoru jedinou instrukcí
 - vytvoří otisk stavu procesoru do paměti a je schopen tento otisk opět načíst (pusha/popa)

Popis procesů

Process Control Block (PCB)

- Obsahuje veškeré údaje o procesu – Linux `task_struct` – `include/linux/sched.h`
- Datová struktura obsahující:
 - Identifikátor procesu (PID) a rodičovského procesu (PPID)
 - Globální stav (process state)
 - Místo pro uložení všech registrů procesoru
 - Informace potřebné pro plánování procesoru/ů
 - Priorita, historie využití CPU
 - Informace potřebné pro správu paměti
 - Informace o právech procesu, kdo ho spustil
 - Stavové informace o V/V (I/O status)
 - Otevřené soubory
 - Proměnné prostředí (environment variables)
 - ... (spousta dalších informací)
 - Ukazatelé pro řazení PCB do front a seznamů

Fronty procesů pro plánování

- Fronta připravených procesů
 - množina procesů připravených k běhu čekajících pouze na přidělení procesoru
- Fronta na dokončení I/O operace
 - samostatná fronta pro každé zařízení
- Seznam odložených procesů
 - množina procesů čekajících na přidělení místa v hlavní paměti, FAP
- Fronty související se synchronizací procesů
 - množiny procesů čekajících synchronizační události
- Fronta na přidělení prostoru v paměti
 - množina procesů potřebujících zvětšit svůj adresní prostor
- ... (další fronty podle potřeb)
- Procesy mezi různými frontami migrují

Obsah

1 Proces

2 Vlákna

3 Od programu k procesu

Program, proces, vlákno

■ Program:

- je soubor (např. na disku) přesně definovaného formátu obsahující
 - instrukce,
 - data
 - údaje potřebné k zavedení do paměti a inicializaci procesu

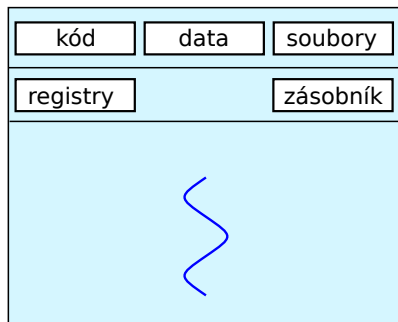
■ Proces:

- je spuštěný program – objekt jádra operačního systému provádějící výpočet podle programu
- je charakterizovaný svým paměťovým prostorem a kontextem (prostor v RAM se přiděluje procesům – nikoli programům!)
- může vlastnit (kontext obsahuje položky pro) otevřené soubory, I/O zařízení a komunikační kanály, které vedou k jiným procesům, ...
- obsahuje jedno či více vláken

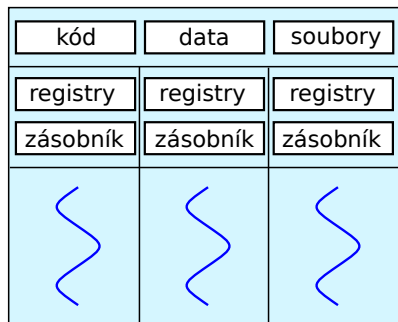
■ Vlákno:

- je sekvence instrukcí vykonávaných procesorem
- sdílí s ostatními vlákny procesu paměťový prostor a další atributy procesu (soubory, ...)
- má vlastní hodnoty registrů CPU

Procesy a vlákna



jednovláknový proces



vícevláknový proces

Vlákno – thread

Vlákno – thread

- Objekt vytvářený v rámci procesu a viditelný uvnitř procesu
- Tradiční proces je proces tvořený jediným vláknem
- Vlákna podléhají plánování a přiděluje se jim strojový čas i procesory
- Vlákno se nachází ve stavech: běží, připravené, čekající, ukončené
- Když vlákno neběží, je kontext vlákna uložený v TCB (Thread Control Block)
 - analogie PCB
 - Linux má stejnou strukturu `task_struct` pro procesy i pro vlákna
 - na informace společné s procesem (např. správa paměti) se vlákno odkazuje k procesu
 - Každý proces je tedy vlastně alespoň jedno vlákno
- Vlákno může přistupovat k globálním proměnným a k ostatním zdrojům svého procesu, data jsou sdílena všemi vlákny stejného procesu
 - Změnu obsahu globálních proměnných procesu vidí všechna ostatní vlákna téhož procesu
 - Soubor otevřený jedním vláknem je viditelný pro všechna ostatní vlákna téhož procesu

Proces

Co patří komu?

kód programu	proces
lokální proměnné	vlákno
globální proměnné	proces
otevřené soubory	proces
zásobník	vlákno
správa paměti	proces
čítač instrukcí	vlákno
registry CPU	vlákno
plánovací stav	vlákno
uživatelská práva	proces

Účel vláken

■ Přednosti

- Vlákno se vytvoří i ukončí rychleji než proces
- Přepínání mezi vlákny je rychlejší než mezi procesy
- Dosáhne se lepší strukturalizace programu

■ Příklady

- Souborový server v LAN
 - Musí vyřizovat během krátké doby několik požadavků na soubory
 - Pro vyřízení každého požadavku se zřídí samostatné vlákno
- Symetrický multiprocessor
 - na různých procesorech mohou běžet vlákna souběžně
- Menu vypisované souběžně se zpracováním prováděným jiným vláknem
 - Překreslování obrazovky souběžně se zpracováním dat
- Paralelizace algoritmu v multiprocessoru

Stavy vláken

- Vlákna podléhají plánování a mají své stavy podobně jako procesy
- Základní stavy
 - běžící
 - připravené
 - čekající
- Všechna vlákna jednoho procesu sdílejí společný adresní prostor
- Vlákna se samostatně neodkládají na disk(process swap), odkládá je jen proces
- Ukončení (havárie) procesu ukončuje všechna vlákna existující v tomto procesu

Možnosti implementace vláken

Vlákna na uživatelské úrovni

- OS zná jenom procesy
- Vlákna vytváří uživatelská knihovna, která střídavě mění spuštěná vlákna procesu
- Pokud jedno vlákno na něco čeká, ostatní vlákna nemohou běžet, protože jádro OS označí jako čekající celý proces
- Pouze staré systémy, nebo jednoduché OS, kde nejsou vlákna potřeba

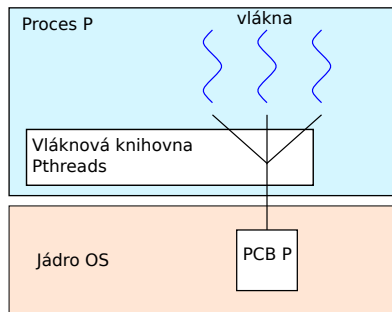
Vlákna na úrovni jádra OS

- Procesy a vlákna jsou plně podporované v jádře
- Moderní operační systémy (Windows, Linux, OSX, Android)
- Vlákno je jednotka plánování činnosti systému

Vlákna na uživatelské úrovni

Problémy vláken na uživatelské úrovni

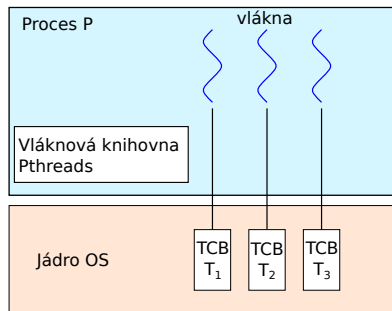
- Jedno vlákno čeká, všechny vlákna čekají
- Proces čeká, ale stav vlákna je běžící
- Dvě vlákna nemohou běžet skutečně paralelně, i když systém obsahuje více CPU



Vlákna v jádře OS

Kernel-Level Threads (KLT)

- Veškerá správa vláken je realizována OS
- Každé vlákno v uživatelském prostoru je zobrazeno na vlákno v jádře (model 1:1)
- JOS vytváří, plánuje a ruší vlákna
- Jádro může plánovat vlákna na různé CPU, skutečný multiprocessing
- Nyní všechny moderní OS: Windows, OSX, Linux, Android



Vlákna v jádře OS

■ Výhody:

- Volání systému neblokuje ostatní vlákna téhož procesu
- Jeden proces může využít více procesorů
- Skutečný paralelismus uvnitř jednoho procesu – každé vlákno běží na jiném procesoru
- Tvorba, rušení a přepínání mezi vlákny je levnější než přepínání mezi procesy
- Netřeba dělat cokoli s přidělenou pamětí

■ Nevýhody:

- Systémová správa je režijně nákladnější než u čistě uživatelských vláken
- Klasické plánování není "spravedlivé": Dostává-li vlákno své časové kvantum, pak procesy s více vlákny dostávají více času
 - Moderní OS ale používají spravedlivé plánování

Pthreads

- Pthreads je knihovna poskytující API pro vytváření a synchronizaci vláken definovaná standardem POSIX.
- Knihovna Pthreads poskytuje unifikované API:
 - Nepodporuje-li JOS vlákna, knihovna Pthreads bude pracovat čistě s ULT
 - Implementuje-li příslušné jádro KLT, pak toho knihovna Pthreads bude využívat
 - Pthreads je tedy systémově závislá knihovna
- Vlákna Linux:
 - Linux nazývá vlákna tasks
 - Linux má stejnou strukturu `task_struct` pro procesy i pro vlákna
 - Lze použít knihovnu `pthread`
 - Vytváření vláken je realizováno službou OS `clone()`

Pthreads API

Příklad: Samostatné vlákno, které počítá součet prvních n celých čísel

```
#include <pthread.h>
#include <stdio.h>

int sum;                                /* sdílená data */
void *runner(void *param);             /* rutina realizující vlákno */

main(int argc, char *argv[]) {
    pthread_t tid;                      /* identifikátor vlákna */
    pthread_attr_t attr;                /* atributy vlákna */
    pthread_attr_init(&attr);           /* inicializuj implicitní atributy */
    pthread_create(&tid, &attr, runner, argv[1]); /* vytvoř vlákno */
    pthread_join(tid, NULL);            /* čekej až vlákno skončí */
    printf("sum = %d\n", sum);
}

void *runner(void *param) {
    int upper = atoi(param);  int i;  sum = 0;
    if (upper > 0) {
        for (i = 1; i <= upper; i++)
            sum += i;
    }
    pthread_exit(0);
}
```

Vlákna ve Windows

- Aplikace ve Windows běží jako proces tvořený jedním nebo více vlákny
- Windows implementují mapování 1:1
- Někteří autoři dokonce tvrdí, že Proces se nemůže vykonávat, neboť je jen kontejnerem pro vlákna a jen ta jsou schopná běhu
- Každé vlákno má:
 - svůj identifikátor vlákna
 - sadu registrů (obsah je ukládán v TCM)
 - samostatný uživatelský a systémový zásobník
 - privátní datovou oblast

Vlákna v Javě

- Vlákna v Javě:
 - Java má třídu „Thread“ a instancí je vlákno
 - Samozřejmě lze z třídy Thread odvodit podtřídu a některé metody přepsat
 - JVM pro každé vlákno vytváří jeho Java zásobník, kde jsou lokální třídy nedostupné pro ostatní vlákna
 - JVM spolu se základními Java třídami vlastně vytváří virtuální stroj obsahující jak vlastní JVM tak i na něm běžící OS podporující vlákna
 - Pokud se jedná o OS podporující vlákna pak jsou vlákna JVM mapována 1:1 na vlákna OS

Vlákna v Javě

Dva příklady jak vytvořit vlákno v Javě

```
class CounterThread extends Thread {  
    public void run() {  
        for(int i = 0; i < 10; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

```
Thread counterThread = new CounterThread();  
  
counterThread.start();
```

```
class Counter implements Runnable {  
    public void run() {  
        for(int i = 0; i < 10; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

```
Runnable counter = new Counter();  
Thread counterThread = new Thread(counter);  
  
counterThread.start();
```

Obsah

1 Proces

2 Vlákna

3 Od programu k procesu

Psaní programů

- Psaní programu je prvním krokem po analýze zadání a volbě algoritmu
- Program zpravidla vytváříme textovým editorem a ukládáme do souboru s příponou indikující programovací jazyk
 - zdroj.c pro jazyk C
 - prog.java pro jazyk Java
 - text.cc, text.cpp pro C++
- Každý takový soubor obsahuje úsek programu označovaný dále jako modul
- V závislosti na typu dalšího zpracování pak tyto moduly podléhají různým sekvencím akcí, na jejichž konci je jeden nebo několik výpočetních procesů
- Rozlišujeme dva základní typy zpracování:
 - Interpretace (bash, python)
 - Kompilace – překlad (C, Pascal)
 - existuje i řada smíšených přístupů (Java – vykonává předkompilovaný a uložený kód, vytvořený překladačem, který je součástí interpretačního systému)

Interpretace

Interpretem rozumíme program, který provádí příkazy napsané v nějakém programovacím jazyku

- vykonává přímo zdrojový kód
 - mnohé skriptovací jazyky a nástroje (např. bash), starší verze BASIC
- překládá zdrojový kód do efektivnější vnitřní reprezentace a tu pak okamžitě „vykonává“
 - jazyky typu Perl, Python, MATLAB apod.

Výhody interpretů:

- rychlý vývoj bez potřeby explicitního překladu a dalších akcí
- nezávislost na cílovém stroji

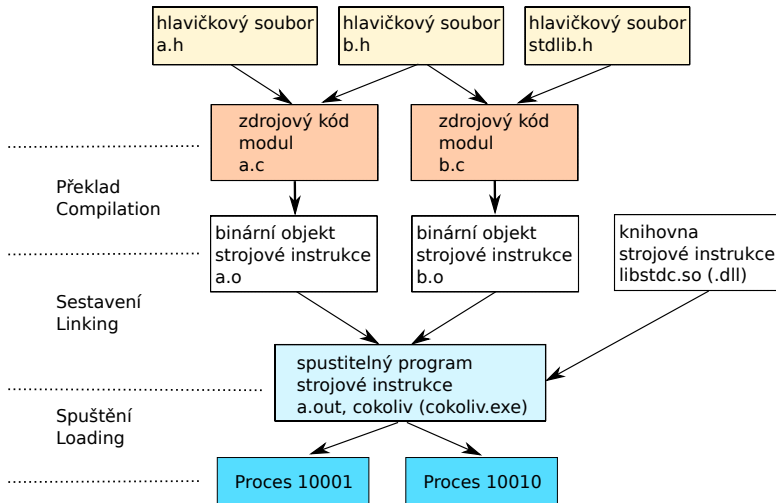
Nevýhody:

- nízká efektivita „běhu programu“
- interpret stále analyzuje zdrojový text (např. v cyklu) nebo se „simuluje“ jiný stroj

Poznámka:

- strojový kód je interpretován hardwarem – CPU

Překlad



Překladač

Úkoly překladače (kompilátoru)

- kontrolovat správnost zdrojového kódu
- „porozumět“ zdrojovému textu programu a převést ho do vhodného „meziproduktu“, který lze dále zpracovávat bez jednoznačné souvislosti se zdrojovým jazykem
- základní výstup kompilátoru bude záviset na jeho typu
 - tzv. nativní překladač generuje kód stroje, na kterém sám pracuje
 - křížový překladač (cross-compiler) vytváří kód pro jinou hardwarovou platformu (např. na PC vyvíjíme program pro vestavěný mikropočítač s procesorem úplně jiné architektury, než má naše PC)
- mnohdy umí překladač generovat i ekvivalentní program v jazyku symbolických adres (assembler)
- častou funkcí překladače je i optimalizace kódu
 - např. dvě po sobě jdoucí čtení téže paměťové lokace jsou zbytečná
 - jde často o velmi pokročilé techniky závislé na cílové architektuře, na zdrojovém jazyku
 - optimalizace je časově náročná, a proto lze úroveň optimalizace volit jako parametr překladu
 - při vývoji algoritmu chceme rychlý překlad, při konečném překladu provozní verze programu žádáme rychlost a efektivitu

Struktura překladače jazyka C

- Předzpracování – preprocessing, vložení souborů a nahrazení maker (`#define`), podmíněný překlad, odstranění komentářů
 - výsledkem je upravený jeden textový soubor pro překlad
- Lexikální analýza
 - výsledek jsou tokeny – rozpoznání stavebních prvků
- Syntaktická a sémantická analýza
 - výsledkem je strom odvození a tabulka symbolů
- Generátor mezikódu
 - výsledkem je abstraktní strojový jazyk – three address code, java – soubory class
- Optimalizace – odstranění redundantních operací, optimalizace cyklů, atp.
 - výsledek optimalizovaný abstraktní kód
- Generátor kódu – přiřazení proměnných registrům
 - výsledkem je binární objekt, který obsahuje strojové instrukce a inicializovaná data

Lexikální analýza

- Lexikální analýza
- převádí textové řetězce na série tokenů (též lexemů), tedy textových elementů detekovaného typu
- např. příkaz: `sedm = 3 + 4` generuje tokeny
 - `(sedm, IDENT)`, `(=, ASSIGN_OP)`, `(3, NUM)`, `(+, OPERATOR)`, `(4, NUM)`
- Již na této úrovni lze detekovat chyby typu „nelegální identifikátor“ (např. `1q`)
- Tvorbu lexikálních analyzátorů lze mechanizovat pomocí programů typu `lex` nebo `flex`

Syntaktická a sémantická analýza

- základem je bezkontextová gramatika
- bezkontextová gramatika je speciálním případem formální gramatiky
- bezkontextová gramatika je čtveřice $G = (V_N, V_T, P, S)$, kde
 - V_N je množina neterminálních symbolů, tj. symbolů které se nevyskytují v popisovaném jazyce
 - V_T je množina terminálních symbolů, tj. symbolů ze kterých je tvořen jazyk, v našem případě jsou terminální symboly výsledkem lexikální analýzy
 - P je množina přepisovacích pravidel, pro bezkontextovou gramatiku jsou pouze pravidla $A \rightarrow \beta$, kde $A \in V_N$ a $\beta \in (V_N \cup V_T)^*$, tzn. β je libovolné slovo složené z terminálů i neterminálů
 - S je počáteční symbol z množiny V_N

Syntaktická a sémantická analýza

Příklad bezkontextová gramatika pro výrazy:

```

expr      : mul_expr
           | expr '+' mul_expr
           | expr '-' mul_expr;

mul_expr  : unary_expr
           | mul_expr '*' unary_expr
           | mul_expr '/' unary_expr;

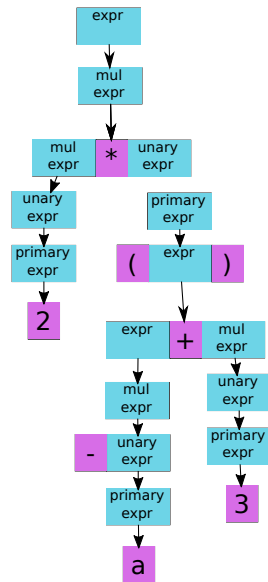
unary_expr : primary_expr
           | unary_op primary_expr;

unary_op   : '&' | '*' | '+' | '-' | '~' | '!';

primary_expr : id | constant | '(' expr ')';
  
```

Odvození je pak již vlastně sémantická analýza (definuje význam věty - programu)

Vpravo vidíte strom odvození výrazu $2*(-a+3)$



Syntaktická a sémantická analýza

- většinou bývá prováděna společným kódem překladače, zvaným parser
- Tvorba parserů se mechanizuje pomocí programů typu yacc či bison
- yacc = Yet Another Compiler Compiler; bison je zvíře vypadající jako yacc
- Programovací jazyky se formálně popisují nejčastěji gramatikami pomocí Extended Backus-Naurovy Formy (EBNF)

```
digit_excluding_zero = "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".
digit                 = "0" | digit_excluding_zero.
natural_number       = digit_excluding_zero,{digit}.
integer              = "0" | ["-"], natural_number.
arit_operator        = "+" | "-" | "*" | "/".
simple_int_expr       = integer,arit_operator,integer.
```

- EBNF pro jazyk C lze nalézt na
http://www.cs.man.ac.uk/~pjj/bnf/c_syntax.bnf
- EBNF pro jazyk Java
<http://cui.unige.ch/isi/bnf/JAVA/AJAVA.html>

Three adress code

- Celý program se popíše trojicemi: operand1 operace operand2
 - Některé operace mají jen jeden operand, druhý je nevyužit, např goto adresa.
- Každá trojice má svoje číslo, které obsahuje výsledek operace
- Nejčastější zápis: **t1:=op1 + op2**

Příklad: $x = \text{sqrt}(a * a - b * b)$

```

t1 := a * a
t2 := b * b
t3 := t1 - t2
t4 := push_param(t3)
t5 := call_sqrt
t6 := x = t5

```

Příklad: for (i=0; i<10; i++) a+=i

```

t1 := i = 0
t2 := goto t7
t3 := a + i
t4 := a = t3
t5 := i + 1
t6 := i = t5
t7 := i < 10
t8 := if t7 goto t3

```


Optimalizace při překladu

Co může překladač optimalizovat

- Elementární optimalizace

- předpočítání konstant

- `n = 1024 * 64` – během překladu se vytvoří konstanta 65536

- znovupoužití vypočtených hodnot

- `if(x**2 + y**2 <= 1) { a = x**2 + y**2; } else { a=0; }`

- detekce a vyloučení nepoužitého kódu

- `if((a>=0) && (a<0)) { never used code; };`

- obvykle se generuje „upozornění“ (warning)

- Sémantické optimalizace

- značně komplikované

- optimalizace cyklů

- lepší využití principu lokality (bude vysvětleno v části virtuální paměti)

- minimalizace skoků v programu – lepší využití instrukční cache

- Celkově mohou být optimalizace velmi náročné během překladu, avšak za běhu programu mimořádně účinné (např. automatická paralelizace)

Generování kódu

- Generátor kódu vytváří vlastní sémantiku "mezikódu"
 - Obecně: Syntaktický a sémantický analyzátor buduje strukturu programu ze zdrojového kódu, zatímco generátor kódu využívá tuto strukturální informaci (např. datové typy) k tvorbě výstupního kódu.
 - Generátor kódu mnohdy dále optimalizuje, zejména při znalosti cílové platformy
 - např.: Má-li cílový procesor více střadačů (datových registrů), dále nepoužívané mezivýsledky se uchovávají v nich a neukládají se do paměti.
- Podle typu překladu generuje různé výstupy
 - assembler (jazyk symbolických adres)
 - absolutní strojový kód
 - pro „jednoduché“ systémy (firmware vestavných systémů)
 - přemístitelný (object) modul
 - speciální kód pro pozdější interpretaci virtuálním strojem
 - např. Java byte-kód pro JVM
- V interpretačních systémech je generátor kódu nahrazen vlastním „interpretem“

Binární objektový modul

Každý objektový modul obsahuje sérii sekcí různých typů a vlastností

- Prakticky všechny formáty objektových modulů obsahují
 - Sekce text obsahuje strojové instrukce a její vlastností je zpravidla EXEC|ALLOC
 - Sekce data slouží k alokaci paměťového prostoru pro inicializovaných proměnných, RW|ALLOC
 - Sekce BSS (Block Started by Symbol) popisuje místo v paměti, které netřeba alokovat ve spustitelném souboru, při start procesu je inicializováno na nulu, RW
- Mnohé formáty objektových modulů obsahují navíc
 - Sekce rodata slouží k alokaci paměťového prostoru konstant, RO|ALLOC
 - Sekci symtab obsahující tabulku globálních symbolů, kterou používá sestavovací program
 - Sekci dynamic obsahující informace pro dynamické sestavení
 - Sekci dynstr obsahující znakové řetězce (jména symbolů pro dynamické sestavení)
 - Sekci dynsym obsahující popisy globálních symbolů pro dynamické sestavení
 - Sekci debug obsahující informace pro symbolický ladící program
 - Detaily viz např.
<http://www.freebsd.org/cgi/man.cgi?query=elf&sektion=5>

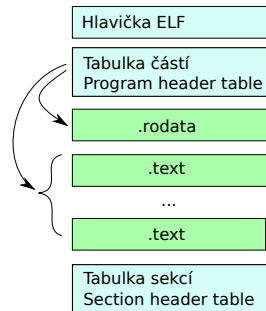
Formáty binárního objektového modulu

- Různé operační systémy používají různé formáty jak objektových modulů tak i spustitelných souborů
- Existuje mnoho různých obecně užívaných konvencí
 - .com, .exe, a .obj
 - formát spustitelných souborů a objektových modulů v MSDOS
 - ELF – Executable and Linkable Format
 - nepoužívanější formát spustitelných souborů, objektových modulů a dynamických knihoven v moderních implementacích POSIX systémů (Linux, Solaris, FreeBSD, NetBSD, OpenBSD, ...). Je též užíván např. i v PlayStation 2, PlayStation 3 a Symbian OS v9 mobilních telefonů.
 - Velmi obecný formát s podporou mnoha platform a způsobů práce s virtuální pamětí, včetně volitelné podpory ladění za běhu
 - Portable Executable (PE)
 - formát spustitelných souborů, objektových modulů a dynamických knihoven (.dll) ve MS-Windows. Označení "portable" poukazuje na univerzalitu formátu pro různé HW platformy, na nichž Windows běží.
 - COFF – Common Object File Format
 - formát spustitelných souborů, objektových modulů a dynamických knihoven v systémech na bázi UNIX V
 - Jako první zavedl sekce s explicitní podporou segmentace a virtuální paměti a obsahuje také sekce pro symbolické ladění

Formát ELF

Formát ELF je shodný pro objektové moduly i pro spustitelné soubory

- ELF Header obsahuje celkové
 - popisné informace
 - např. identifikace cílového stroje a OS
 - typ souboru (obj vs. exec)
 - počet a velikosti sekcí
 - odkaz na tabulku sekcí
 - ...
- Pro spustitelné soubory je podstatný seznam sekcí i modulů.
- Pro sestavování musí být moduly popsány svými sekcemi.
- Sekce jsou příslušných typů a obsahují „strojový kód“ či data
- Tabulka sekcí popisuje jejich typ, alokační a přístupové informace a další údaje potřebné pro práci sestavovacího či zaváděcího programu



ELF struktura

Soubor NOVA kern/include/elf.h

```
class Eh {
public:
    uint32 ei_magic;
    uint8 ei_class, ei_data,
        ei_version, ei_pad[9];
    uint16 type, machine;
    uint32 version;
    mword entry, ph_offset,
        sh_offset;
    uint32 flags;
    uint16 eh_size, ph_size, ph_count,
        sh_size, sh_count, strtabs;
};

enum {
    PF_X = 0x1,
    PF_W = 0x2,
    PF_R = 0x4,
};
```

```
class Ph {
public:
    enum {
        PT_NULL      = 0,
        PT_LOAD       = 1,
        PT_DYNAMIC    = 2,
        PT_INTERP     = 3,
        PT_NOTE       = 4,
        PT_SHLIB      = 5,
        PT_PHDR       = 6,
    };
    uint32 type;
    uint32 f_offs;
    uint32 v_addr;
    uint32 p_addr;
    uint32 f_size;
    uint32 m_size;
    uint32 flags;
    uint32 align;
};
```

ELF struktura použití

```

int f = open(argv[1], O_RDONLY), i;
unsigned char buf[2048];
if (f>=0) {
    read(f, buf, sizeof(Eh));
    Eh *eh=(Eh *)buf;
    printf("Magic %08x - %c%c%c%c\n",eh->ei_magic,eh->ei_magic&0xff,
        (eh->ei_magic>>8)&0xff,(eh->ei_magic>>16)&0xff,(eh->ei_magic>>24)&0xff);
    printf("Entry point %08lx\n", eh->entry);
    printf("Program headers Num=%i, offset=%lu size Eh %i\n", eh->ph_count,
        eh->ph_offset, sizeof(Ph));

    int num = eh->ph_count;
    int read_num = eh->ph_offset+num*32;
    read(f, buf+sizeof(Eh), read_num-sizeof(Eh));

    for (i=0; i<num; i++) {
        Ph *ph=(Ph *)&buf[eh->ph_offset+i*32];
        printf("Program header n%i: type %i f_offs %08x, v_addr %08x, f_size %04x,
            flags %0x, align %i\n",
            i, ph->type & 0xff, ph->f_offs, ph->v_addr, ph->f_size, ph->flags, ph->align);
    }
}

```

Sestavování a externí symboly

- V objektovém modulu jsou (aspoň z hlediska sestavování) potlačeny lokální symboly (např. lokální proměnné uvnitř funkcí – jsou nahrazeny svými adresami, symbolický tvar má smysl jen pro případné ladění - debugging)
- globální symboly slouží pro vazby mezi moduly a jsou 2 typů
 - exportované symboly – jsou v příslušném modulu plně definovány, je známo jejich jméno a je známa i sekce, v níž se symbol vyskytuje a relativní adresa symbolu vůči počátku sekce.
 - importované symboly – symboly z cizích modulů, o kterých je známo jen jejich jméno, případně typ sekce, v níž by se symbol měl nacházet (např. pro odlišení, zda symbol představuje jméno funkce či jméno proměnné)

Sestavování a externí symboly

Příklad z materiálů k přednáškám.

Soubor a.h:

```
extern int i_a;
```

```
int f_a(short int x);
```

Soubor b.c:

```
#include "a.h"
```

```
int i_b;
```

```
int f_b(int x) {
    i_b = f_a((short)x);
    i_a = (x/2);
    return (x>>16);
}
```

Tabulka symbolů (zkráceno)

Symbol table '.symtab' contains 12 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
8:	000004	4	OBJECT	GLOBAL	DEFAULT	COM	i_b
9:	000000	52	FUNC	GLOBAL	DEFAULT	1	f_b
10:	000000	0	NOTYPE	GLOBAL	DEFAULT	UND	f_a
11:	000000	0	NOTYPE	GLOBAL	DEFAULT	UND	i_a

Sekce modulu b.o (readelf -a b.o)

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	0000	0000	0000	00		0	0	0
[1]	.text	PROGBITS	0000	0034	0034	00	AX	0	0	1
[2]	.rel.text	REL	0000	01a4	0018	08	I	9	1	4
[3]	.data	PROGBITS	0000	0068	0000	00	WA	0	0	1
[4]	.bss	NOBITS	0000	0068	0000	00	WA	0	0	1
[5]	.comment	PROGBITS	0000	0068	002a	01	MS	0	0	1
[6]	.note.GNU-stack									
		PROGBITS	0000	0092	0000	00		0	0	1
[7]	.eh_frame	PROGBITS	0000	0094	0038	00	A	0	0	4
[8]	.rel.eh_frame									
		REL	0000	01bc	0008	08	I	9	7	4
[9]	.symtab	SYMTAB	0000	00cc	00c0	10		10	8	4
[10]	.strtab	STRTAB	0000	018c	0015	00		0	0	1
[11]	.shstrtab	STRTAB	0000	01c4	0057	00		0	0	1

Sestavování a externí symboly

Překlad funkce f_b

Modul b.o

00000000 <f_b>:

```

0: 55          push    %ebp
1: 89 e5       mov     %esp,%ebp
3: 83 ec 08    sub     $0x8,%esp
6: 8b 45 08    mov     0x8(%ebp),%eax
9: 98          cwtl
a: 83 ec 0c    sub     $0xc,%esp
d: 50          push    %eax
e: e8 fc ff ff call    f <f_b+0xf>
13: 83 c4 10    add     $0x10,%esp
16: a3 00 00 00 mov     %eax,0x0
1b: 8b 45 08    mov     0x8(%ebp),%eax
1e: 89 c2       mov     %eax,%edx
20: c1 ea 1f    shr     $0x1f,%edx
23: 01 d0       add     %edx,%eax
25: d1 f8       sar     %eax
27: a3 00 00 00 mov     %eax,0x0
2c: 8b 45 08    mov     0x8(%ebp),%eax
2f: c1 f8 10    sar     $0x10,%eax
32: c9          leave
33: c3          ret

```

Relokační sekce b.o (readelf -a b.o)

Relocation section '.rel.text' at

offset 0x1a4 contains 3 entries:

Offset	Info	Type	Sym.Value	Sym. Name
0000000f	00000a02	R_386_PC32	00000000	f_a
00000017	00000801	R_386_32	00000004	i_b
00000028	00000b01	R_386_32	00000000	i_a

- Relokační tabulka musí obsahovat odkazy na symboly jejichž poloha není známa v době překladu (f_a, i_a)
- Relokační tabulka potřebuje i známý symbol i_b, protože ve výsledném programu může být na jiném místě

Sestavování a externí symboly

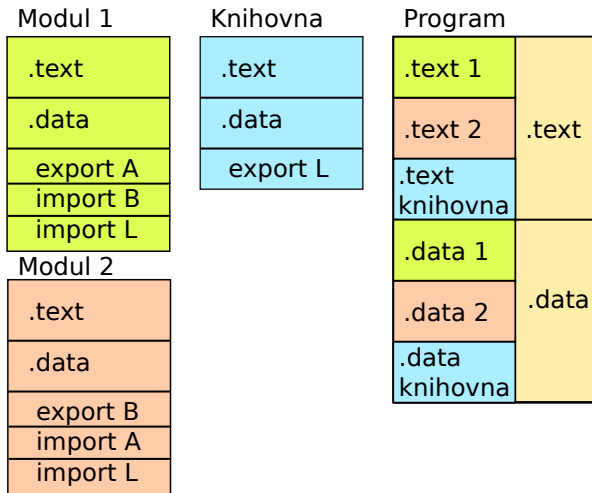
Funkce f_b uvnitř výsledného programu
Před sestavením

```
00000000 <f_b>:
 0: 55          push    %ebp
 1: 89 e5       mov     %esp,%ebp
 3: 83 ec 08    sub     $0x8,%esp
 6: 8b 45 08    mov     0x8(%ebp),%eax
 9: 98          cwtl
 a: 83 ec 0c    sub     $0xc,%esp
 d: 50          push    %eax
 e: e8 fc ff ff call    f <f_b+0xf>
13: 83 c4 10    add     $0x10,%esp
16: a3 00 00 00 mov     %eax,0x0
1b: 8b 45 08    mov     0x8(%ebp),%eax
1e: 89 c2       mov     %eax,%edx
20: c1 ea 1f    shr     $0x1f,%edx
23: 01 d0       add     %edx,%eax
25: d1 f8       sar     %eax
27: a3 00 00 00 mov     %eax,0x0
2c: 8b 45 08    mov     0x8(%ebp),%eax
2f: c1 f8 10    sar     $0x10,%eax
32: c9          leave  %eax
33: c3          ret
```

Po vytvoření programu

```
0000057f <f_b>:
57f: 55          push    %ebp
580: 89 e5       mov     %esp,%ebp
582: 83 ec 08    sub     $0x8,%esp
585: 8b 45 08    mov     0x8(%ebp),%eax
588: 98          cwtl
589: 83 ec 0c    sub     $0xc,%esp
58c: 50          push    %eax
58d: e8 8b ff ff call    51d <f_a>
592: 83 c4 10    add     $0x10,%esp
595: a3 10 20 00 mov     %eax,0x2010
59a: 8b 45 08    mov     0x8(%ebp),%eax
59d: 89 c2       mov     %eax,%edx
59f: c1 ea 1f    shr     $0x1f,%edx
5a2: 01 d0       add     %edx,%eax
5a4: d1 f8       sar     %eax
5a6: a3 0c 20 00 mov     %eax,0x200c
5ab: 8b 45 08    mov     0x8(%ebp),%eax
5ae: c1 f8 10    sar     $0x10,%eax
5b1: c9          leave  %eax
5b2: c3          ret
```

Externí symboly



Statické knihovny

- Knihovna je vlastně balík binárních objektových modulů
- Podle symbolů požadovaných moduly programu se hledají moduly v knihovnách, které tyto symboly exportují
- Nový modul z knihovny může vyžadovat další symboly z dalších modulů
- Pokud po projití všech modulů programu i všech modulů knihovny není symbol nalezen, je ohlášena chyba a nelze sestavit výsledný program

Dynamické knihovny

- Sestavovací program pracuje podobně jako při sestavování statickém, ale dynamické knihovny nepřidává do výsledného programu.
- Odkazy na symboly z dynamických knihoven je nutné vyřešit až při běhu programu. Existují v zásadě dva přístupy:
 - Vyřešení odkazů **při zavádění programu** do paměti – všechny nepropojené symboly extern se propojí před spuštěním programu
 - **Opožděné sestavování** – na místě nevyřešených odkazů připojí sestavovací program malé kousky kódu (zvané stub), které zavolají systém, aby odkaz vyřešil. Při běhu pak „stub“ zavolá operační systém, který zkontroluje, zda potřebná dynamická knihovna je v paměti (není-li zavede ji do paměti počítače), ve virtuální paměti knihovnu připojí tak, aby ji proces viděl. Následně stub nahradí správným odkazem do paměti a tento odkaz provede.
 - Výhodné z hlediska využití paměti, neboť se nezavádí knihovny, které nebudou potřeba.

Dynamické knihovny PLT/GOT

V předcházejícím případě jsme použili dynamickou knihovnu glibc.
Začátek programu (zkráceno)

```
080482f0 <_start>:
80482f0:      31 ed                xor     %ebp,%ebp
...
8048307:      68 03 84 04 08      push   $0x8048403
804830c:      e8 cf ff ff ff      call   80482e0 <__libc_start_main@plt>
```

Funkce `__libc_start_main@plt` je zodpovědná za dynamickou funkci `start_main` z knihovny `libc`

```
080482e0 <__libc_start_main@plt>:
80482e0:      ff 25 10 a0 04 08    jmp     *0x804a010
80482e6:      68 08 00 00 00      push   $0x8
80482eb:      e9 d0 ff ff ff      jmp     80482c0 <_init+0x2c>
```

V okamžiku kompilace a spuštění je v paměti na adrese `0x804a010` hodnota `0x80482e6`

```
Disassembly of section .got.plt:
0804a000 <_GLOBAL_OFFSET_TABLE_>:
...
804a010:      e6 82 04 08
```

- Po zavedení knihovny dojde k přepsání GOT položky pro tuto funkci na správnou adresu funkce v paměti
- Při dalším volání tedy již instrukce `jmp *0x804a010` skočí přímo do funkce `__libc_start_main`

PIC a DLL

Dynamické knihovny jsou sdíleny různými procesy. Buď musí být na stejné pozici/relokovatelná (dll) nebo musí být na pozici nezávislé (PIC)

■ PIC = Position Independent Code

- Překladač generuje kód nezávislý na umístění v paměti
- Skoky v kódu a odkazy na data jsou buď relativní vůči IP, nebo podle GOT – Global offset table
- Pokud nelze k adresaci použít registr IP, je nutné zjistit svoji polohu v paměti:

```

        call .tmp1
.tmp1:  pop %edi
        addl $_GLOBAL_OFFSET_TABLE - .tmp1, %edi

```

- pro x86_64 lze použít pro adresaci registr RIP (číslo 0x2009db je posunutí GOT od prováděné instrukce, spočítáno překladačem)

```

mov    $0x2009db(%rip), %rax

```

- kód je sice obvykle delší, avšak netřeba cokoli modifikovat při sestavování či zavádění
- užívá se zejména pro dynamické knihovny
- v poslední době se využívá i pro programy

DLL

- DLL – knihovna je na stejném místě pro všechny procesy
 - pokud se zavádí nová knihovna pro další process, pak musí být na volném místě
 - pokud není volné místo tam, kam je připravena, musí se relokovat – posunout všechny vnitřní pevné odkazy (skoky a data)
 - MS přiděluje místa v paměti na požádání vývojářů, aby minimalizoval možnost kolize
 - Vaše knihovna bude pravděpodobně v kolizi a bude se proto přesouvat – pomalejší provedení programu s touto knihovnou

Zavaděč

- Zavaděč – loader – je zodpovědný za spuštění programu
- V POSIX systémech je to vlastně obsluha služby „execve“
- Úkoly zavaděče
 - vytvoření „obrazu procesu“ (memory image) v odkládacím prostoru na disku a částečně i v hlavní paměti v závislosti na strategii virtualizace, případné vyřešení nedefinovaných odkazů
 - sekce ze spustitelného souboru se stávají segmenty procesu (pokud správa paměti nepodporuje segmentaci, pak stránkami)
 - segmenty získávají příslušná „práva“ (RW, RO, EXEC, ...)
 - inicializace „registrů procesu“ v PCB
 - např. ukazatel zásobníku a čítač instrukcí
 - předání řízení na vstupní adresu procesu