B4B35OSY: Operační systémy

Lekce 3. Procesy a vlákna

Petr Štěpán
stepan@fel.cvut.cz

October 16, 2017

Outline

Outline

Proces

- Výpočetní proces (job, task) – spuštěný program
- Proces je identifikovatelný jednoznačným číslem
  - PID – Process Identification Digit
- Stav procesu lze v každém okamžiku jeho existence jednoznačně určit
  - přidělené zdroje; události, na něž proces čeká; prioritu;
- 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 odpovědi procesu prokládáním běhů procesů
- Přidělovat procesům požadované systémové prostředky
  - Soubory, V/V zařízení, synchronizační prostředky
- Podporovat 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 POSIXu)
  - 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 POSIXu je každý proces potomkem jiného procesu
  - Výjimka: proces init vytvořen při spuštění systému
  - Spustí řadu sh skriptů (rc), ty inicializují celý systém a vytvoří démony (procesy běžící na pozadí bez úplného kontextu) service ve Win32
  - init spustí pro terminály proces getty, který čeká na uživatele $\rightarrow$ login $\rightarrow$ uživatelův shell

POSIX vytvoření procesu

Rodič vytváří nový proces – potomka – voláním služby fork()

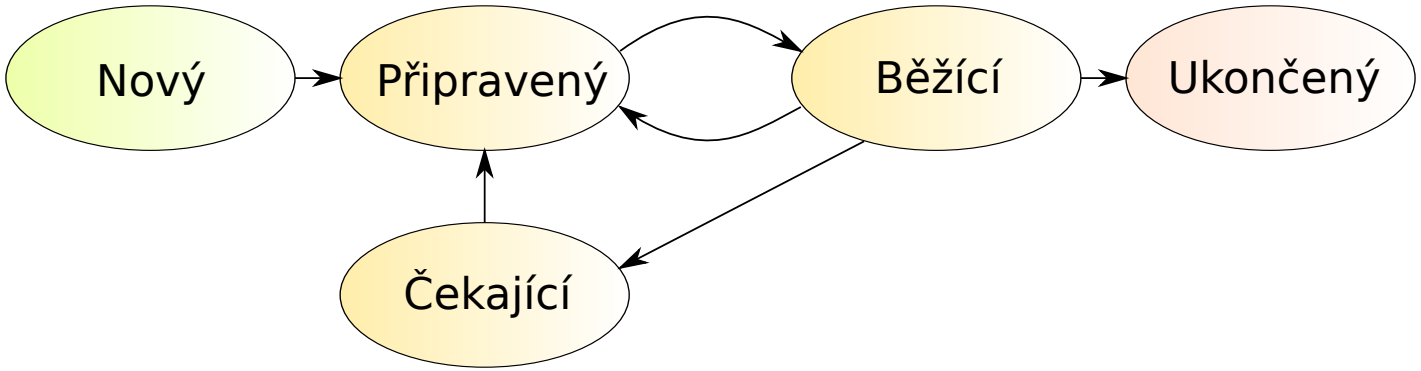
- vznikne identická kopie rodičovského procesu až na:
  - návratovou hodnotu systémového volání
  - hodnotu PID, PPID – číslo rodičovského procesu
- návratová hodnota určuje, kdo je potomek a kdo rodič
  - 0 – jsem potomek
  - PID – jsem rodič a získávám PID potomka
- do adresního prostoru potomka se automaticky zavádí 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

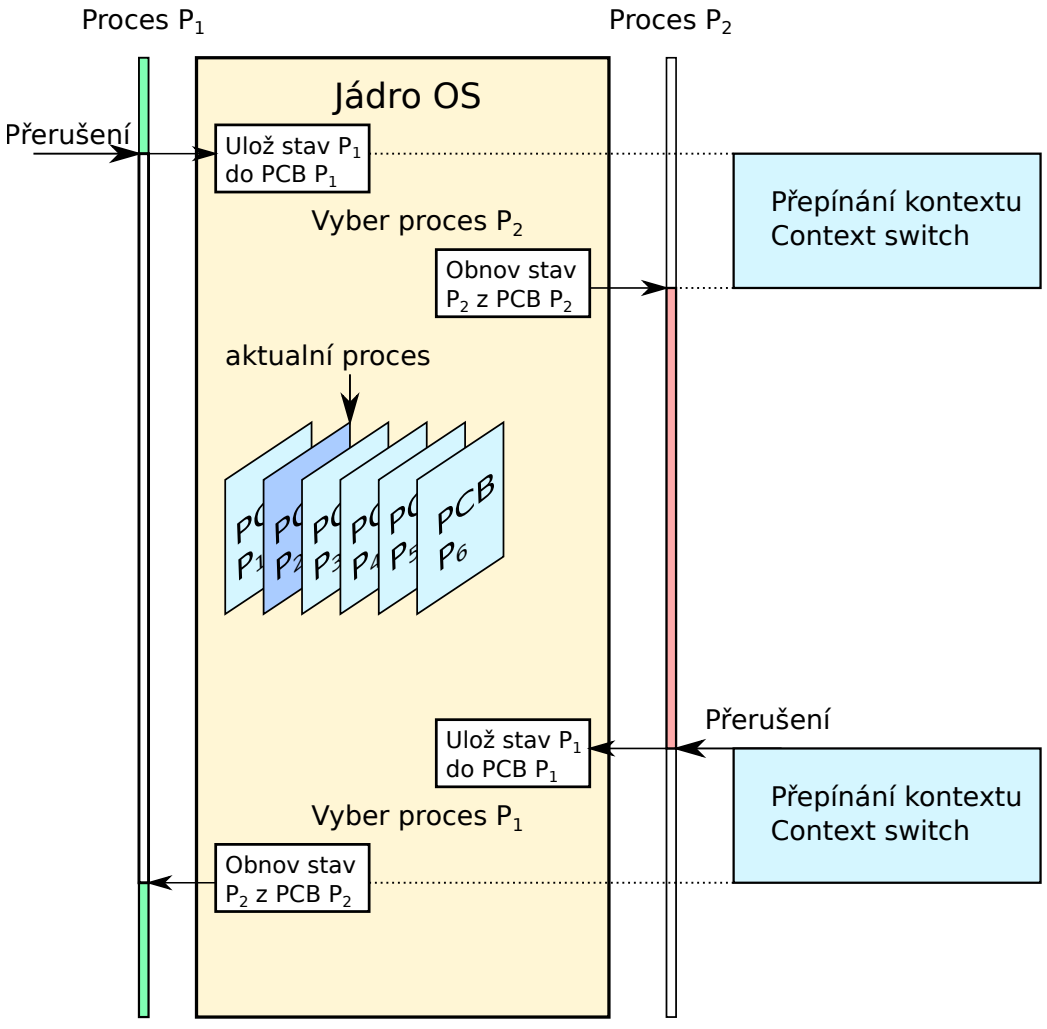
- Proces provede poslední příkaz programu a žádá OS o ukončení voláním služby exit(status)
  - Stavová data procesu-potomka (status) se musí předat procesu-rodiči, který čeká v provádění služby wait()
  - Zdroje končícího procesu jádro uvolní
- Proces může skončit také:
  - přílišným nárokem na paměť (požadované množství paměti není a nebude k dispozici)
  - narušením ochrany paměti („zabloudění“ programu)
  - pokusem o provedení nedovolené (privilegované) operace (zakázaný přístup k systémovému prostředku, r/o soubor)
  - aritmetickou chybou (dělení nulou, arcsin(2), ...) či neopravitelnou chybou V/V
  - žádostí rodičovského procesu (v POSIXu signál)
  - zánikem rodiče
    - 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 na běh, 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

- Přechod 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 $PCBP_1$
  - jádru OS rozhodne, který proces poběží dál – P_2
  - Obnoví se stav procesu P_2 z PCB P_2
- Přepnutí kontextu představuje režijní ztrátu (zátěž systému)
  - 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)
  - $\cdots$ (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

Outline

Program, proces, vlákno

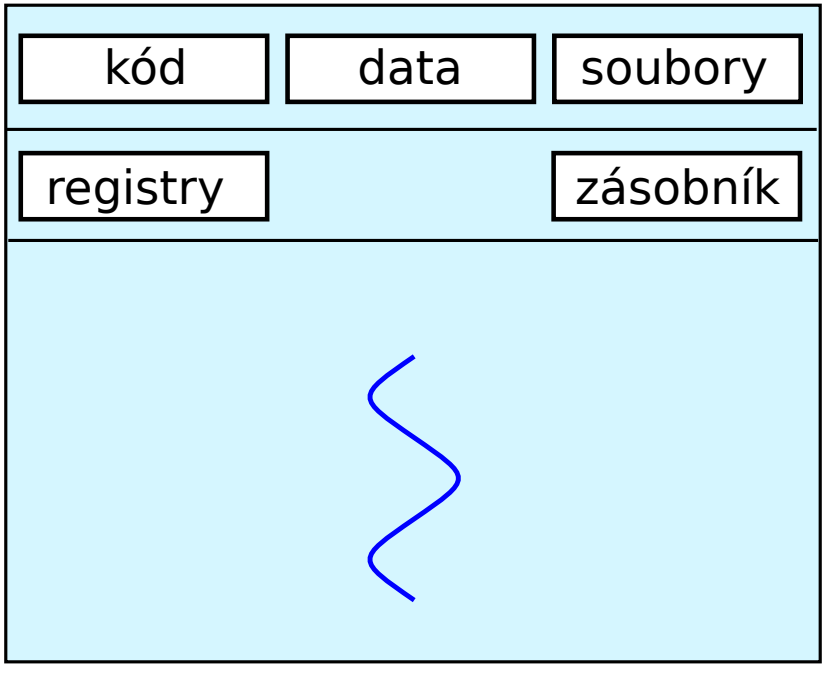
- Program je:
  - soubor přesně definovaného formátu obsahující
    - instrukce,
    - data
    - údaje potřebné k zavedení do paměti a inicializaci procesu
- Proces je:
  - spuštěný program – objekt jádra operačního systému provádějící výpočet podle programu
  - charakterizovaný svým paměťovým prostorem a kontextem (prostor v RAM se přiděluje procesům – nikoli programům!)
  - může vlastnit soubory, I/O zařízení a komunikační kanály, které vedou k jiným procesům
- Vlákno je:
  - objekt, který vykonává stejný program v rámci procesu
  - sdílí s procesem a ostatními vlákny procesu společný paměťový prostor

Vlákno - thread

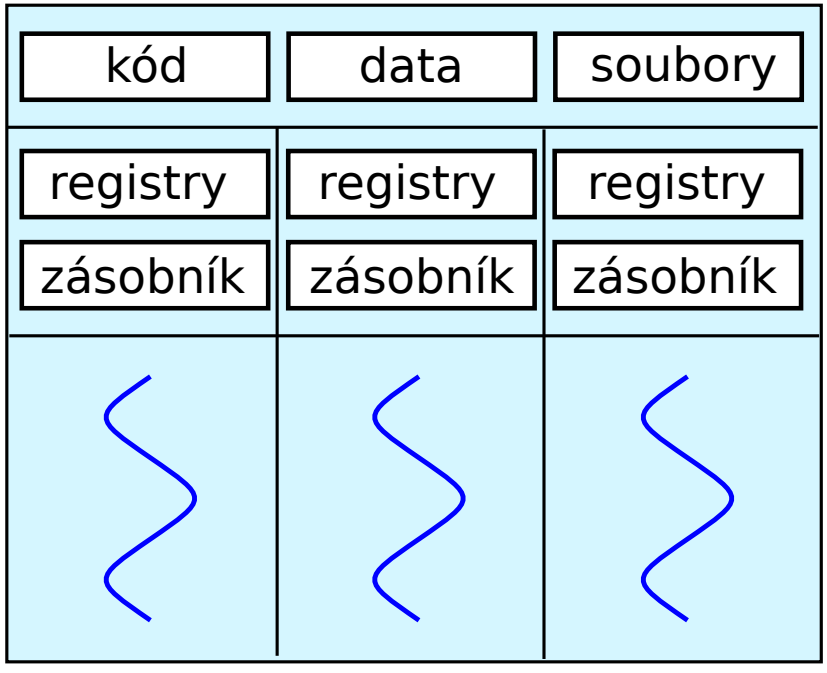
Vlákno – thread

- 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
  - Každý proces je tedy vlastně alespoň jedno vlákno
- Vlákno může přistupovat k datům a k ostatním zdrojům svého procesu, data jsou sdíleny všemi vlákny stejného procesu
- Změnu obsahu některých dat procesu vidí všechna ostatní vlákna téhož procesu
- Soubor otevřený jedním vláknem je viditelný pro všechna ostatní vlákna téhož procesu

Procesy a vlákna



jednovláknový proces



vícevláknový proces

Proces

Co patří komu?

kód programu	počítač
lokální proměnné	vlákno
globální data	proces
systémové zdroje	proces
zásobník	vlákno
správa paměti	proces
čítač instrukcí	vlákno
registry CPU	vlákno
plánovací stav	vlákno
uživatelská práva	proces
změna uživatele	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 multiprocesor
    - na různých procesorech mohou běžet vlákna souběžně
  - Menu vypisované souběžně se zpracováním prováděným jiným vláknem
    - Překreslování obrazovky souběžně se zpracováním dat
  - Paralelizace algoritmu v multiprocesoru

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, odkládá je jen proces
- Ukončení (havárie) procesu ukončuje všechna vlákna existující v tomto procesu

OS a vlákna

Vlákna na uživatelské úrovni

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

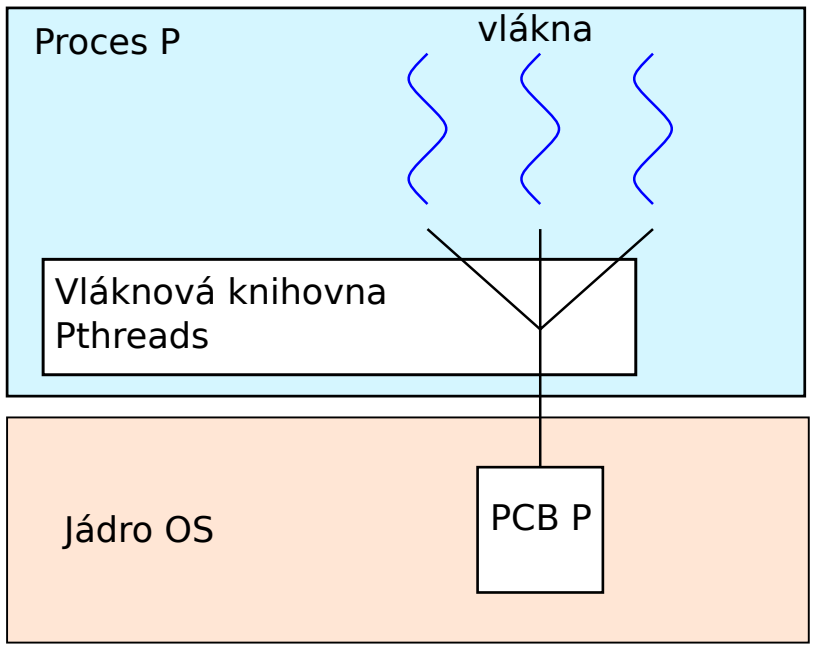
Vlákna na úrovni jádra OS

- 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

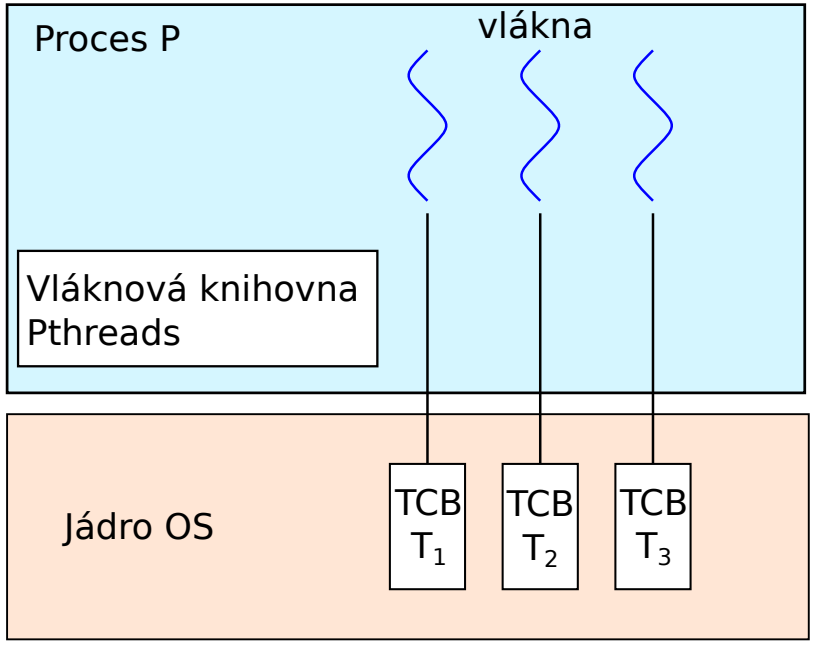
- Jedno vlákno čeká, všechny vlákna čekají
- Proces čeká, ale stav vlákna je běžící
- Dvě vlákna nemohou běžet skutečně paralelně, i když systém obsahuje více CPU



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 cokoliv s přidělenou pamětí
  - I moduly jádra mohou mít vícevláknový charakter
- 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

Pthreads

- Pthreads je POSIX-ový standard definující API pro vytváření a synchronizaci vláken a specifikace chování těchto vláken
- 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 pthreads
  - 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

TODO

- 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 „hardware“ (vlastní JVM) tak i na něm běžící OS podporující vlákna
  - Většinou 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();
```

Outline

Psaní programů

- Při 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 akcích
- nezávislost na cílovém stroji

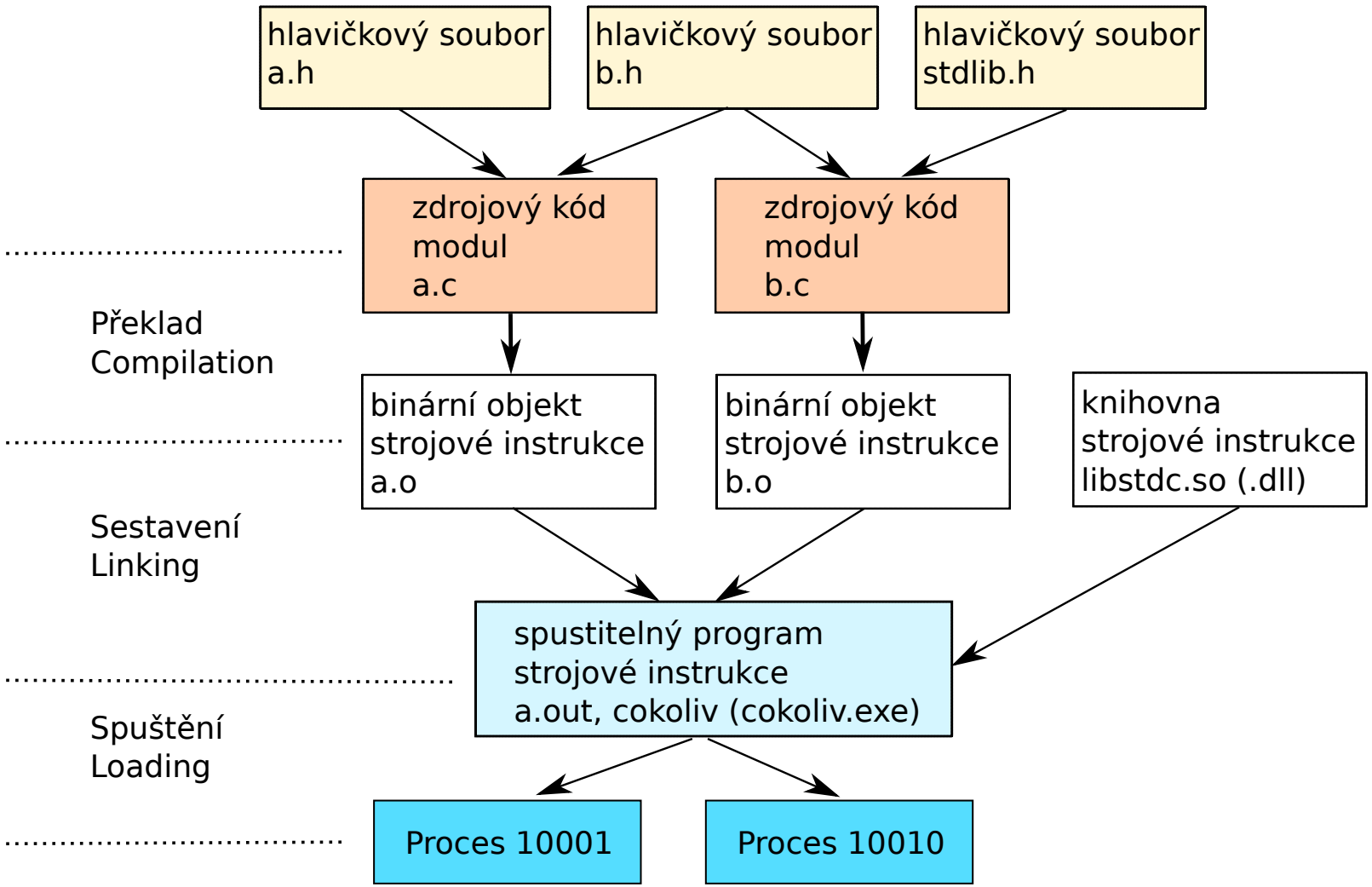
Nevýhody:

- 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


hlavičkový soubor
a.h
hlavičkový soubor
b.h
hlavičkový soubor
stdlib.h
zdrojový kód
modul
a.c
zdrojový kód
modul
b.c
Překlad
Compilation
binární objekt
strojové instrukce
a.o
binární objekt
strojové instrukce
b.o
knihovna
strojové instrukce
libstdc.so (.dll)
Sestavení
Linking
spustitelný program
strojové instrukce
a.out, cokoliv (cokoliv.exe)
Spuštění
Loading
Proces 10001
Proces 10010

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 úrovně optimalizace volit jako parametr překladu
  - při vývoji algoritmu chceme rychlý překlad, při konečném překladu provozní verze programu žádáme rychlost a efektivitu

Struktura překladače

- Předzpracování - preprocesing, vložení souborů a nahrazení maker (#define), podmíněný překlad
  - výsledkem je upravený text 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, pro javu soubory class
- Optimalizace - odstranění redundantních operací, optimalizace cyklů, atp.
  - výsedek 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

- 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: $x = sqrt(a^2 - b^2)$

```
t1 := a * a
t2 := b * b
t3 := t1 - t2
t4 := sqrt(t3)
t5 := x = t4
```

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 (viz téma 8)
  - 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 střádačů (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“
  - ukážeme několik principů používaných interprety (a někdy i generátory cílového kódu)

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, 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 ladicí program
  - Detaily viz např. http://www.freebsd.org/cgi/man.cgi?query=elf&sektion=5

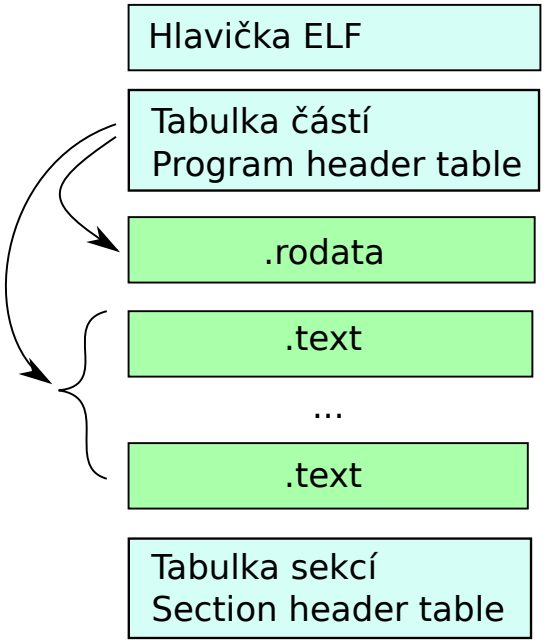
Formáty binárního objektového modulu

- 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
    - nejpoužívanější 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 platforem 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

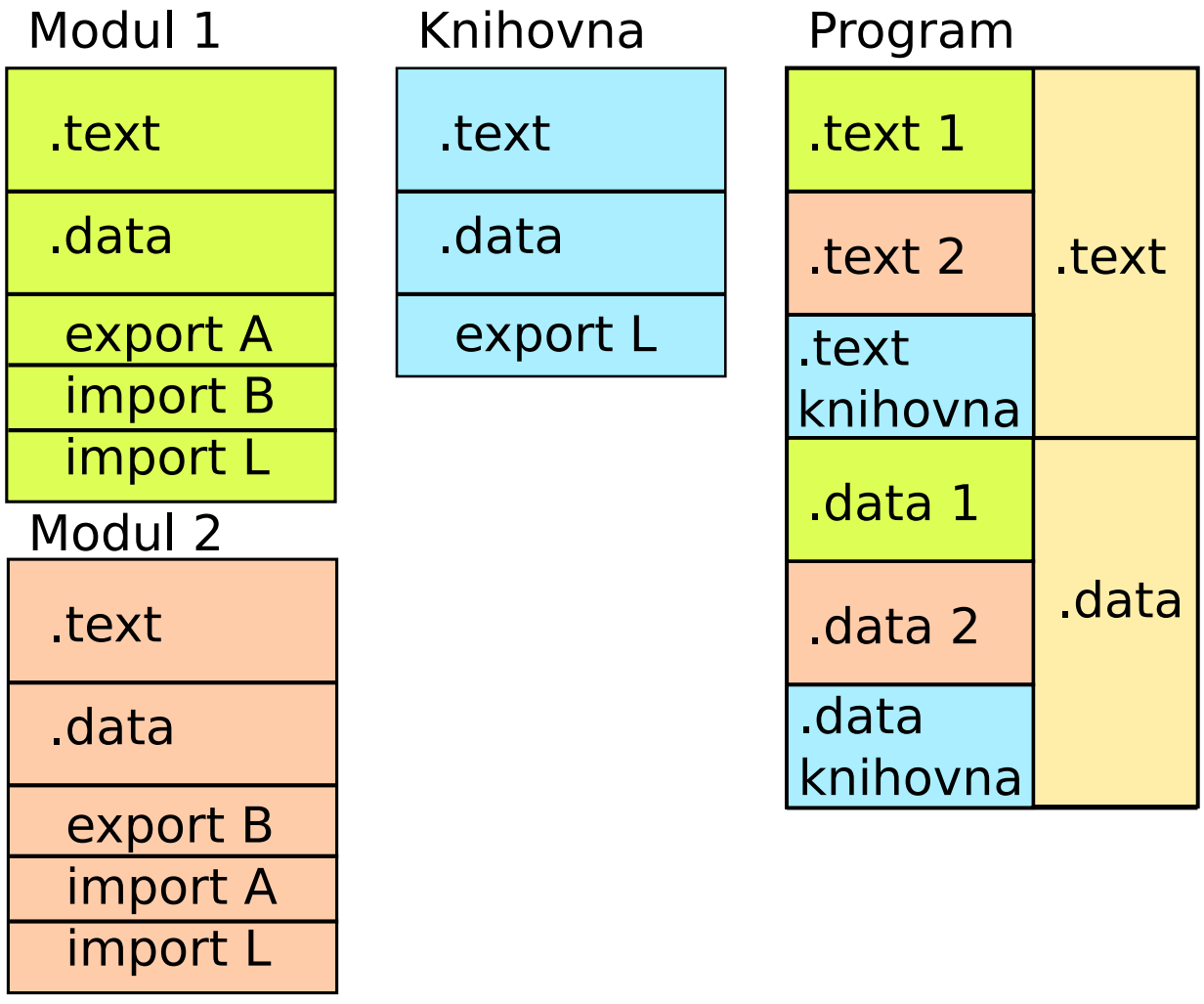
- ELF Header obsahuje celkové
  - popisné informace
  - např. identifikace cílového stroje a OS
  - typ souboru (obj vs. exec)
  - počet a velikosti sekci
  - odkaz na tabulku sekcí
  - ...
- Pro spustitelné soubory je podstatný seznam sekcí i modulů.
- Pro sestavování musí být moduly popsány svými sekcemi.
- 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



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)
- 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)

Externí symboly



Statické knihovny

- Knihovna je vlastně mnoho 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
  - 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.

PIC a DLL

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

- PIC = Position Independent Code
  - Překladač generuje kód nezávislý na umístění v paměti
  - skoky v kódu jsou buď relativní, nebo podle GOT - Global offsett table
  - Pokud nelze k adresaci použít registr ip, je nutné zjistit svoji polohu v paměti:

```
        call .tmp1
.tmp1:  pop %ebx
        addl $_GLOBAL_OFFSET_TABLE - .tmp1, %ebx
```

  - kód je sice obvykle delší, avšak netřeba cokoliv modifikovat při sestavování či zavádění
  - užívá se zejména pro dynamické knihovny

DLL

- DLL - knihovna je ne 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