B4B35OSY: Operační systémy

Lekce 3. Procesy a vlákna

Petr Štěpán
stepan@fel.cvut.cz

6. října, 2022

Outline

Obsah

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 akcích
- nezávislost na cílovém stroji

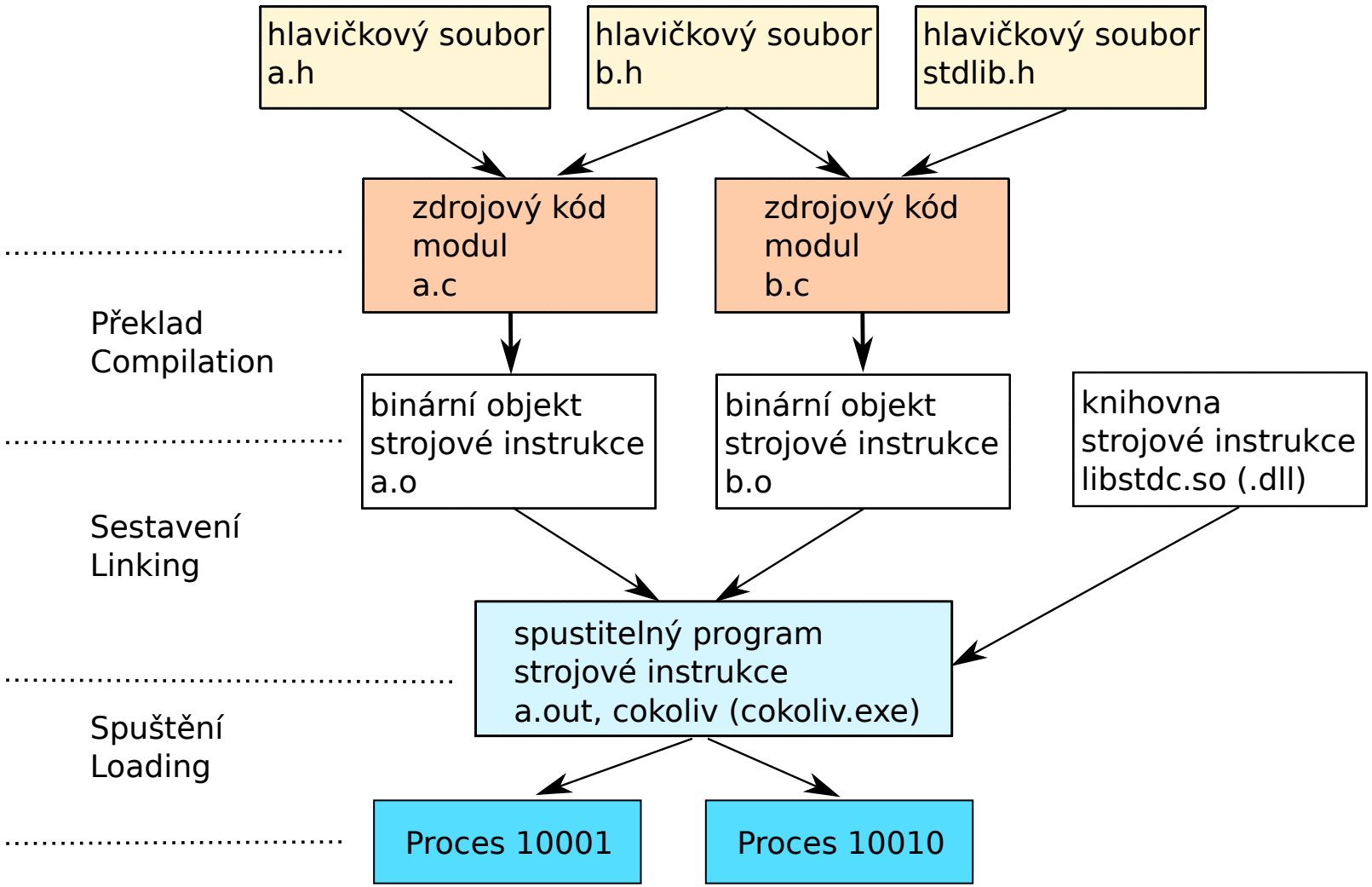
Nevýhody:

- 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 jazyka C

- Předzpracování – preprocesing, 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

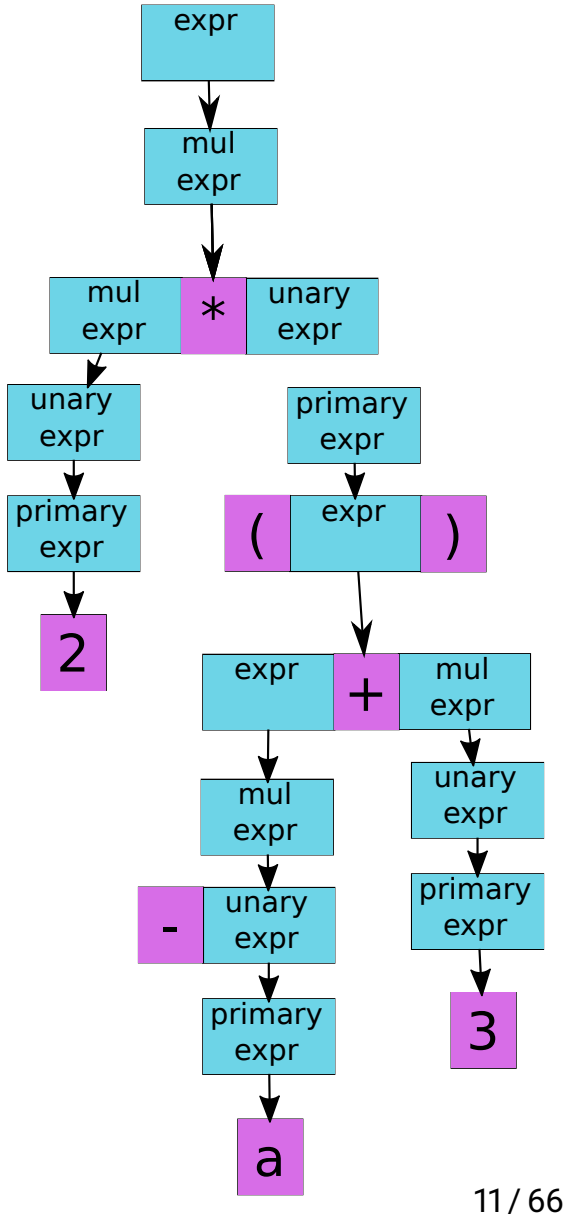
- základem je bezkontextová gramaticka
- bezkontextová gramatika je speciálním případem formální gramatiky
- bezkontextová gramatika je čtveřice $G = (V_N, V_T, P, S)$, kde
  - V_N je množina neterminálních symbolů, tj. symbolů které se nevyskytují v popisovaném jazyce
  - V_T je množina terminálních symbolů, tj. symbolů ze kterých je tvořen jazyk, v případě překladače jsou terminální symboly lexemy, které jsou výsledkem lexikání 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 jak (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: $x = sqrt(a * a - b * b)$

```
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 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“

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

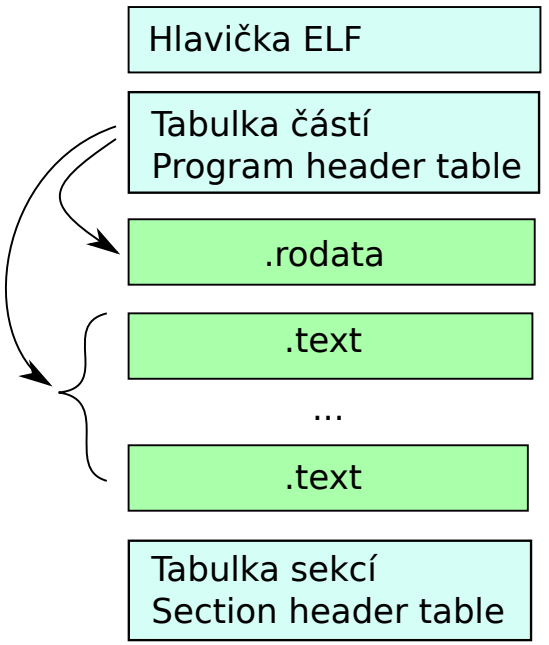
Formáty binárního objektového modulu

- 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



ELF struktura

Systém NOVA používá formát ELF, proto obsahuje definici jeho hlavičky 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, strtab;
};

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);
  }
}
```

Kvíz

Je nutné uvést deklaraci funkce z cizího modulu?

A - Není to nutné

B - Není to nutné, ale mělo by se to dělat kvůli přehlednosti kódu

C - Je to nutné, aby překladač věděl, že jsme zadali správně jméno funkce

D - Je to nutné, aby překladač správně vygeneroval kód pro volání funkce

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            00000000 000000 000000 00      0   0  0
[ 1] .text             PROGBITS        00000000 000034 000034 00  AX  0   0  1
[ 2] .rel.text         REL             00000000 0001a4 000018 08   I  9   1  4
[ 3] .data             PROGBITS        00000000 000068 000000 00  WA  0   0  1
[ 4] .bss              NOBITS          00000000 000068 000000 00  WA  0   0  1
[ 5] .comment          PROGBITS        00000000 000068 00002a 01  MS  0   0  1
[ 6] .note.GNU-stack
                       PROGBITS        00000000 000092 000000 00      0   0  1
[ 7] .eh_frame         PROGBITS        00000000 000094 000038 00   A  0   0  4
[ 8] .rel.eh_frame
                       REL             00000000 0001bc 000008 08   I  9   7  4
[ 9] .symtab           SYMTAB          00000000 0000cc 0000c0 10     10   8  4
[10] .strtab           STRTAB          00000000 00018c 000015 00      0   0  1
[11] .shstrtab         STRTAB          00000000 0001c4 000057 00      0   0  1
```

Sestavování a externí symboly

Modul `b.o`

Překlad funkce `f_b`

```
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 ff call   f <f_b+0xf>
13: 83 c4 10        add    $0x10,%esp
16: a3 00 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 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 známá 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 ff        call   f <f_b+0xf>
  13: 83 c4 10              add    $0x10,%esp
  16: a3 00 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 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
```

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 ff        call   51d <f_a>
 592: 83 c4 10              add    $0x10,%esp
 595: a3 10 20 00 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 00        mov    %eax,0x200c
 5ab: 8b 45 08              mov    0x8(%ebp),%eax
 5ae: c1 f8 10              sar    $0x10,%eax
 5b1: c9                    leave
 5b2: c3                    ret
```

Kvíz

Jaký je rozdíl mezi statickou knihovnou a dynamickou knihovnou?

A - Žádný rozdíl není

B - Obě knihovny jsou součástí programu, ale dynamická může změnit místo uložení v uživatelském prostoru

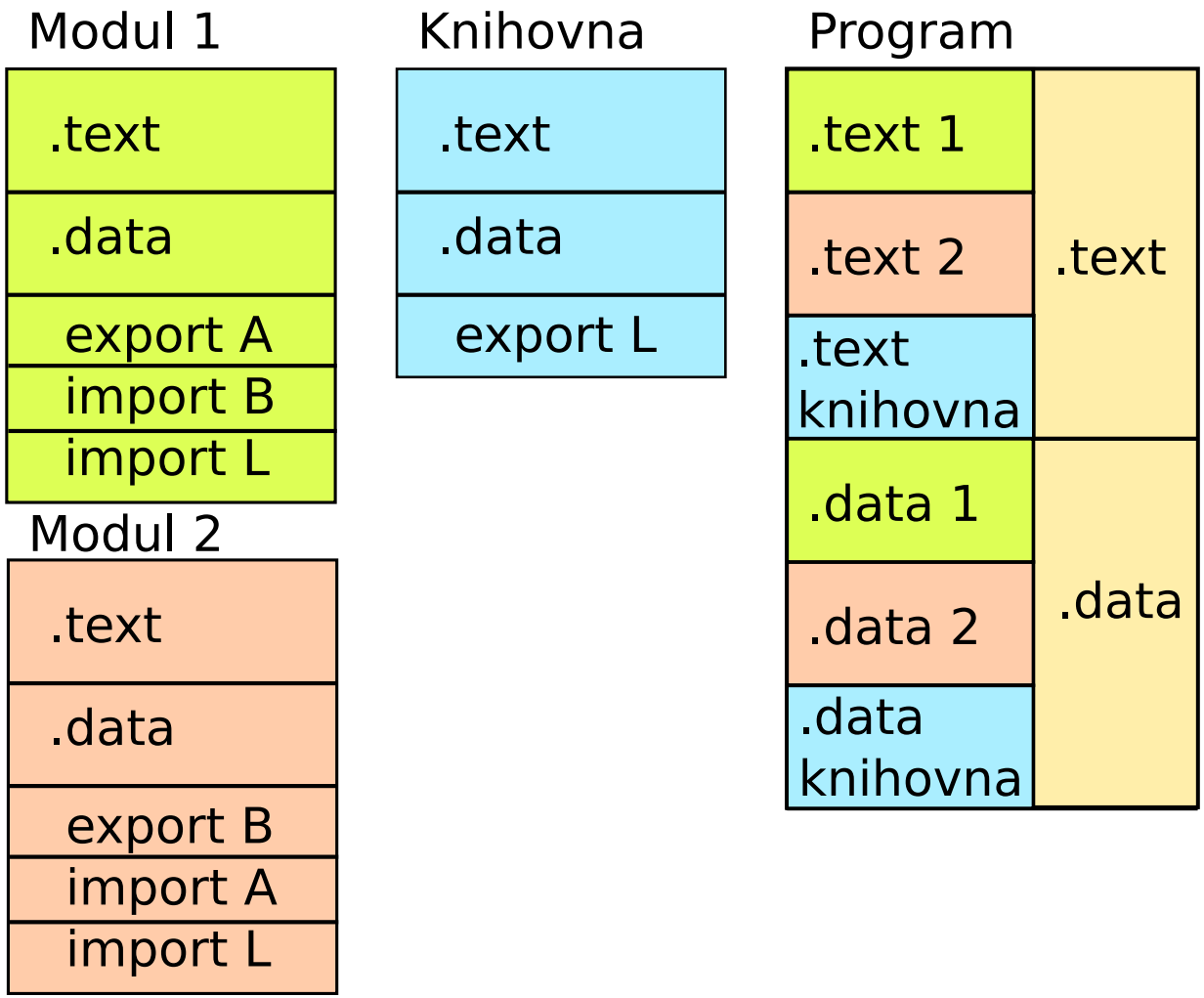
C - Statická knihovna je součástí programu, dynamická knihovna se stane součástí programu až při využití funkce z této knihovny

D - Statická knihovna je součástí programu, dynamická knihovna je součástí jádra OS

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

Externí symboly - statická knihovna



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 dynaminckou 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 knihovna 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 cokoliv 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

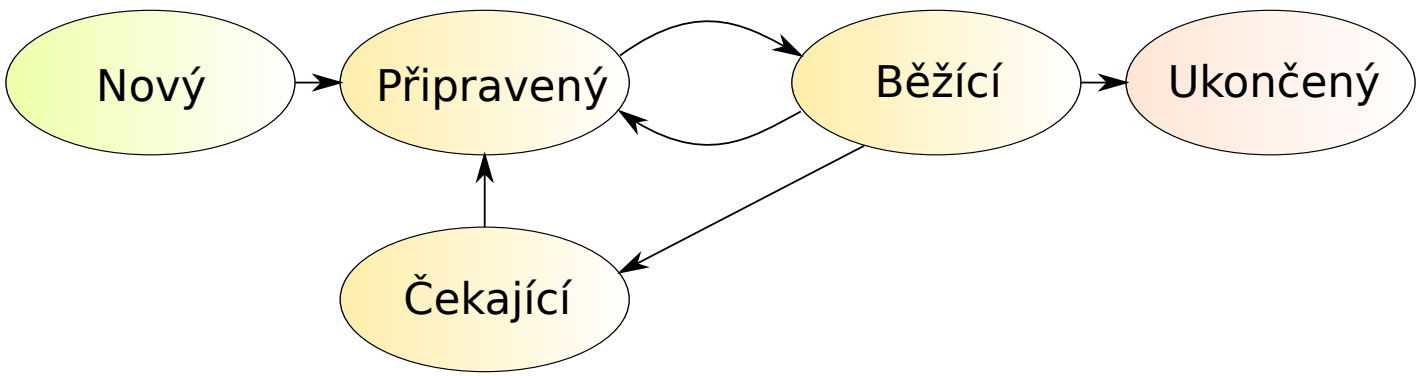
- Zavaděč – loader – je zodpovědný za spuštění programu
- V POSIX systémech je to vlastně obsluha služby „execve“
- Úkoly zavaděče
  - vytvoření „obrazu procesu“ (memory image) v odkládacím prostoru na disku a částečně i v hlavní paměti v závislosti na strategii virtualizace, případné vyřešení nedefinovaných 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

Obsah

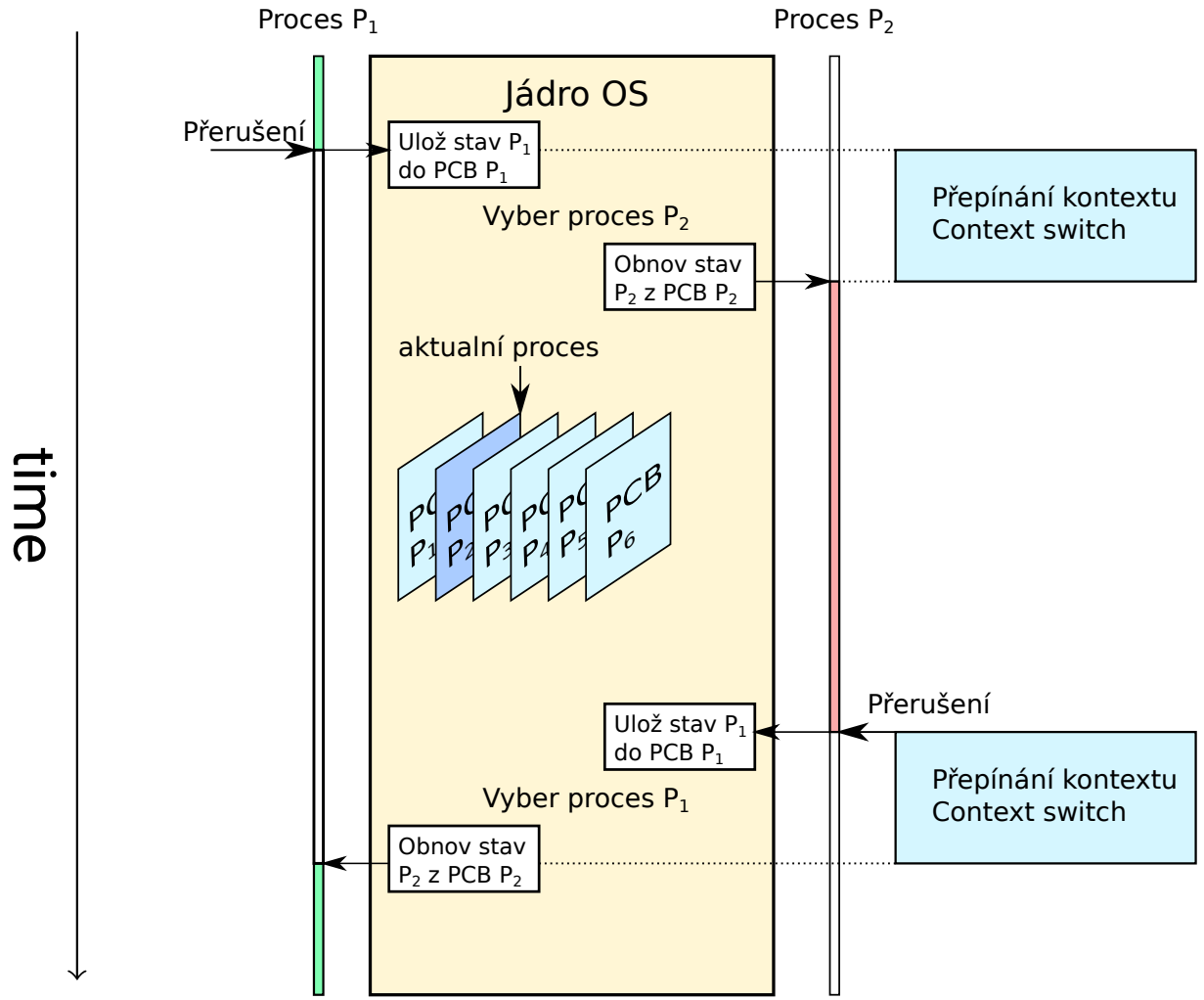
Stavy procesu

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 jsou 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
  - 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)

NOVA – přepínání procesů – vstup do jádra

Vstup do jádra

- Prohlédněte si `kern/src/entry.S`

```
;; System-Call Entry
.align 4, 0x90
.globl entry_sysenter
entry_sysenter:
    cld
    pop   %esp
    lea   -44(%esp), %esp
    pusha
    mov   $(KSTCK_ADDR + PAGE_SIZE), %esp
    jmp   syscall_handler
```

- pop esp - nastav zásobník podle hodnoty ze systémového zásobníku
- pusha - ulož všechny registry do struktury uchovávající data procesu
- mov $(KSTCK_ADDR + PAGE_SIZE), %esp - nastav stack pro volání funkcí uvnitř obsluhy systémového volání
- jmp syscall_handler - spusť funkci syscall_handler z `kern/src/ec_syscall.cc`

NOVA – přepínání procesů – návrat z jádra

Návrat z jádra

- Prohlédněte si `kern/src/ec.cc`

```
void Ec::ret_user_sysexit() {
    asm volatile (
      "lea %0, %%esp;"
      "popa;"
      "sti;"
      "sysexit"
      : : "m" (current->regs)
      : "memory");
    UNREACHED;
}
```

- lea %0, esp - nastav zásobník na current->regs
- popa - obnov všechny registry
- sti - povol přerušení
- sysexit - vrať se ze systémového volání

Popis procesů

Process Control Block (PCB), v NOVĚ třída Ec – Execution context

- 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

Druhy plánovačů

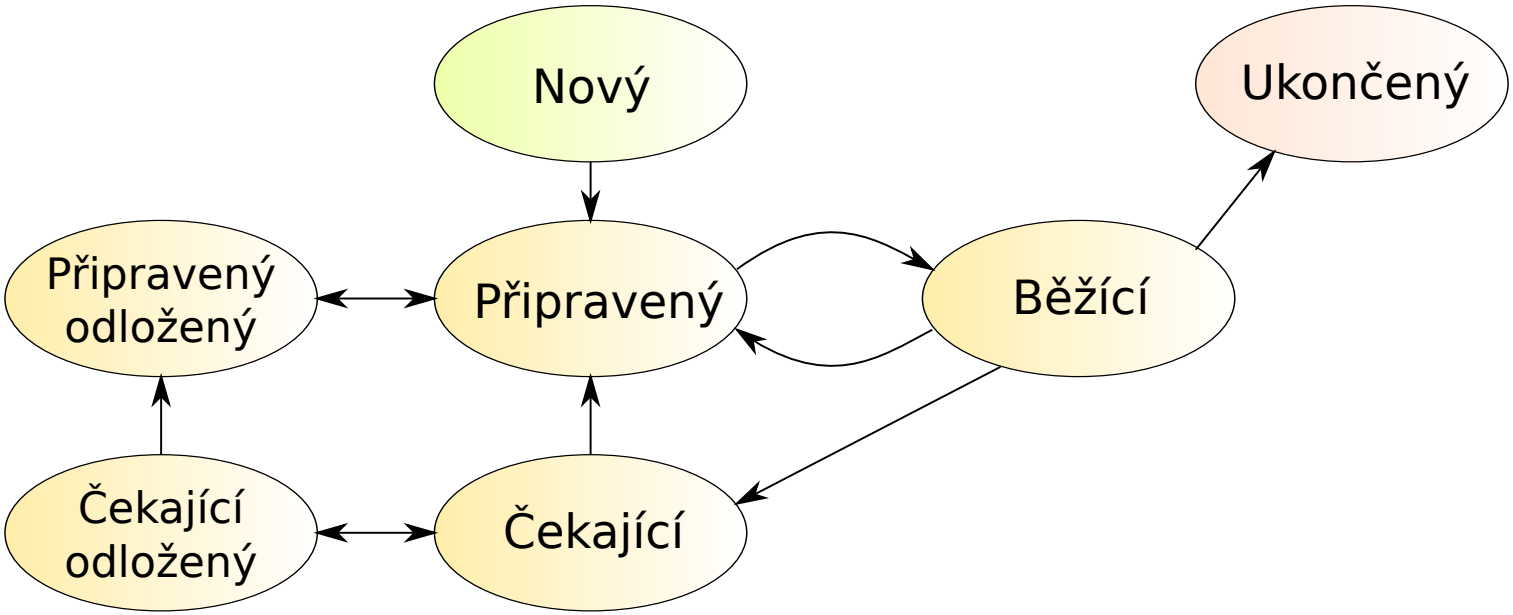
- Krátkodobý plánovač (operační plánovač, dispečer):
  - Základní správa procesoru/ů
  - Vybírá proces, který poběží na uvolněném procesoru přiděluje procesu procesor (CPU)
  - vyvoláván velmi často, musí být extrémně rychlý
- Střednědobý plánovač (taktický plánovač)
  - Úzce spolupracuje se správou hlavní paměti
  - Taktika využívání omezené kapacity fyzické paměti při multitaskingu
  - Vybírá, který proces je možno zařadit mezi odložené procesy
  - uvolní tím prostor zabíraný procesem v fyzické paměti
  - Vybírá, kterému odloženému procesu lze znovu přidělit prostor v paměti počítače
- Dlouhodobý plánovač (strategický plánovač, job scheduler)
  - Vybírá, který požadavek na výpočet lze zařadit mezi procesy, a definuje tak stupeň multiprogramování
  - Je volán zřídka (jednotky až desítky sekund), nemusí být rychlý
  - V interaktivních systémech se používá velmi omezeně, např. plánování aktualizací

Stavy procesu

Nové stavy spojené s odkládáním procesu na disk při nedostatku fyzické paměti:

- Odložený připravený
- Odložený čekající

Moderní OS většinou neprovádí odkládání celých procesů, ale při nedostatku paměti pak hrozí thrashing (podrobněji probereme při stránkování).



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

Dispečer

- Dispečer pracuje s procesy, které jsou v hlavní paměti a jsou schopné běhu, tj. připravené (ready)
- Existují 2 typy plánování
  - nepreemptivní plánování (kooperativní plánování, někdy také plánování bez předbíhání)
    - běžícímu procesu nelze „násilně“ odejmout CPU, proces se musí procesoru vzdát, nebo ho nabídnout
    - historické operační systémy, kdy nebyla od systému podpora preempce
    - nyní se používá zpravidla jen v „uzavřených systémech“, kde jsou předem známy všechny procesy a jejich vlastnosti. Navíc musí být naprogramovány tak, aby samy uvolňovaly procesor ve prospěch procesů ostatních
  - preemptivní plánování (plánování s předbíháním),
- procesu schopnému dalšího běhu může být procesor odňat i „bez jeho souhlasu“, tedy kdykoliv
- plánovač rozhoduje v okamžiku:
  1. kdy některý proces přechází ze stavu běžící do stavu čekající nebo končí
  2. kdy některý proces přechází ze stavu čekající do stavu připravený
  3. přijde vnější podnět od HW prostřednictvím přerušení, nejčastěji od časovače
- První případ se vyskytuje v obou typech plánování
- Další dva jsou použity pouze pro plánování preemptivní

Kritéria plánování

Kritéria plánování

- Uživatelsky orientovaná
  - čas odezvy
    - doba od vzniku požadavku do reakce na něj
  - doba obrátky
    - doba od vzniku procesu do jeho dokončení
  - konečná lhůta (deadline)
    - požadavek dodržení stanoveného času dokončení
  - předvídatelnost
    - Úloha by měla být dokončena za zhruba stejnou dobu bez ohledu na celkovou zátěž systému
    - Je-li systém vytížen, prodloužení odezvy by mělo být rovnoměrně rozděleno mezi procesy
- Systémově orientovaná
  - průchodnost
    - počet procesů dokončených za jednotku času
  - využití procesoru
    - relativní čas procesoru věnovaný aplikačním procesům
  - spravedlivost
    - každý proces by měl dostat svůj čas (ne „hladovění“ či „stárnutí“)
  - vyvažování zátěže systémových prostředků
    - systémové prostředky (periferie, hlavní paměť) by měly být zatěžovány v čase rovnoměrně

Základní plánovače

Ukážeme plánování:

- FCFS (First-Come First-Served)
- SPN (SJF) (Shortest Process Next)
- SRT (Shortest Remaining Time)
- cyklické (Round-Robin)
- zpětnovazební (Feedback)

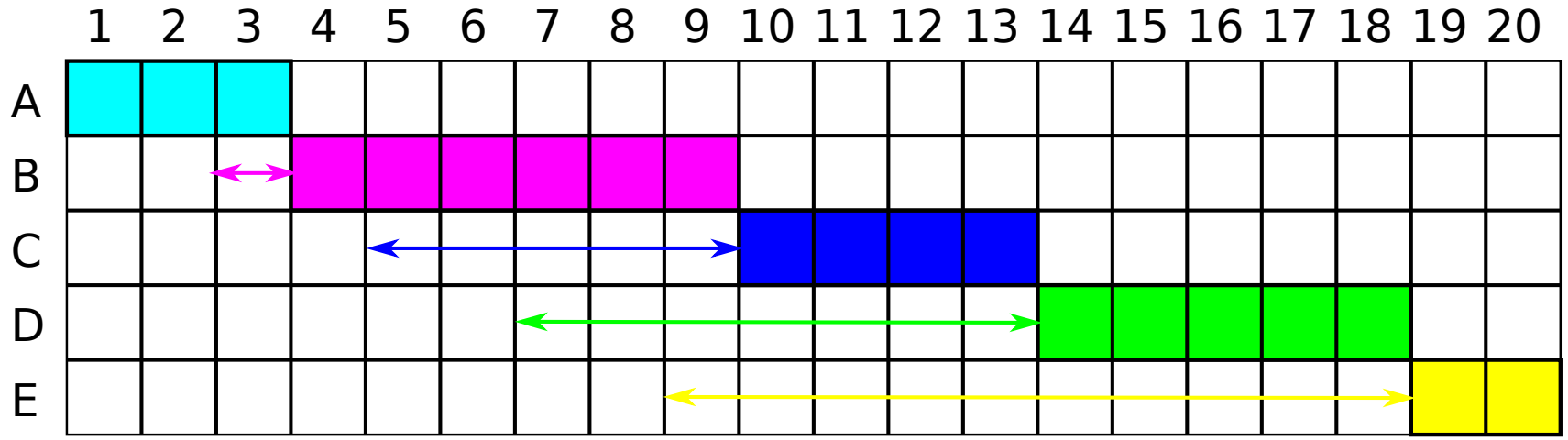
Příklad pro ilustraci algoritmů:

Proces	Čas příchodu	Potřebný čas
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

FCFS

- FCFS = First Come First Served – prostá fronta FIFO
- Nejjednodušší nepreemptivní plánování
- Nově příchozí proces se zařadí na konec fronty
- Průměrné čekání může být velmi dlouhé

Příklad:



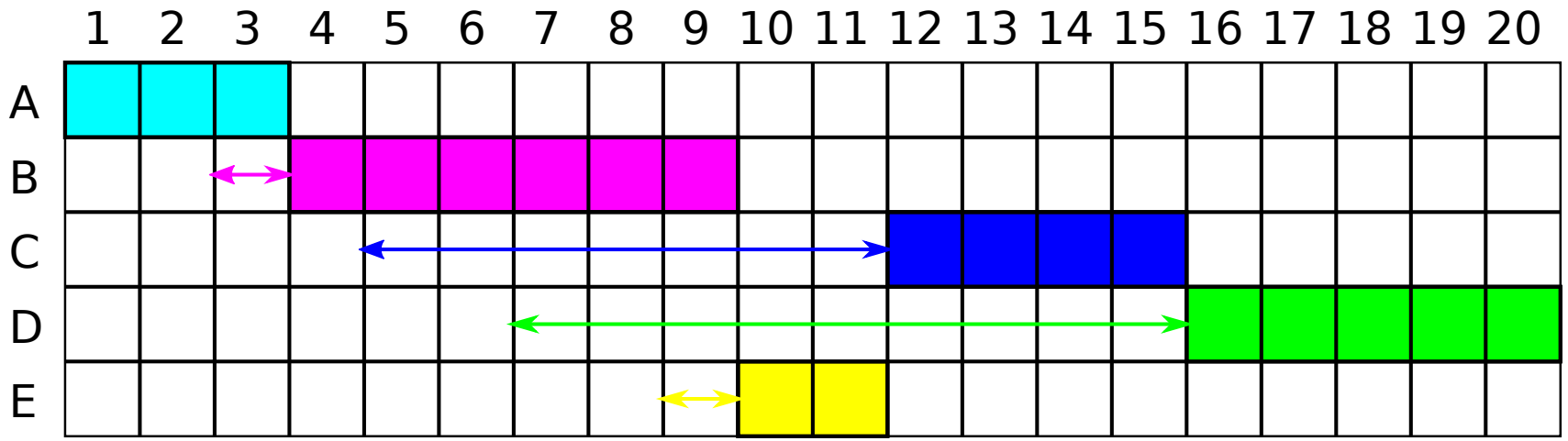
- Průměrné čekání $T_{Avg} = \frac{0+1+5+7+10}{5} = 4.6$
- Průměrné čekání bychom mohli zredukovat, pokud by proces E běžel hned po B.

FCFS – vlastnosti

- FCFS je primitivní nepreemptivní plánovací postup
- Průměrná doba čekání T_{Avg} silně závisí na pořadí přicházejících dávek
- Krátké procesy, které se připravily po dlouhém procesu, vytváří tzv. konvojový efekt
  - Všechny procesy čekají, až skončí dlouhý proces
- Pro krátkodobé plánování se FCFS samostatně fakticky nepoužívá.
  - Používá se pouze jako složka složitějších plánovacích postupů

SPN

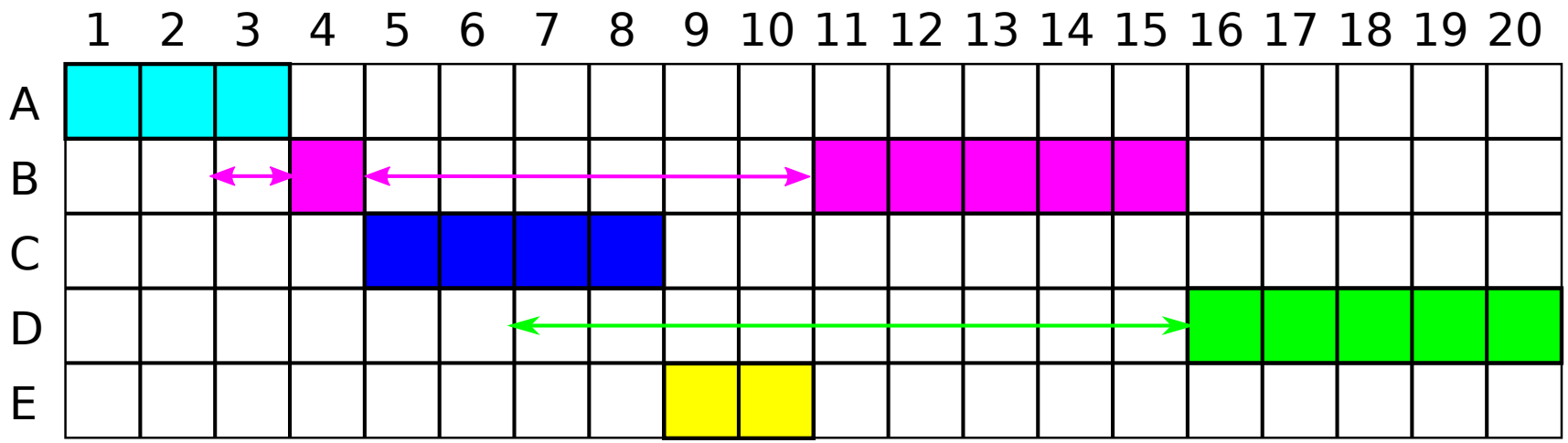
- SPN = Shortest Process Next (nejkratší proces jako příští); též nazýváno SJF = Shortest Job First
  - Opět nepreemptivní
  - Vybírá se připravený proces s nejkratší příští dávkou CPU
  - Krátké procesy předbíhají delší, nebezpečí stárnutí dlouhých procesů
  - Je-li kritériem kvality plánování průměrná doba čekání, je SPN optimálním algoritmem, což se dá exaktně dokázat



- Průměrné čekání $T_{Avg} = \frac{0+1+7+9+1}{5} = 3.6$

SRT

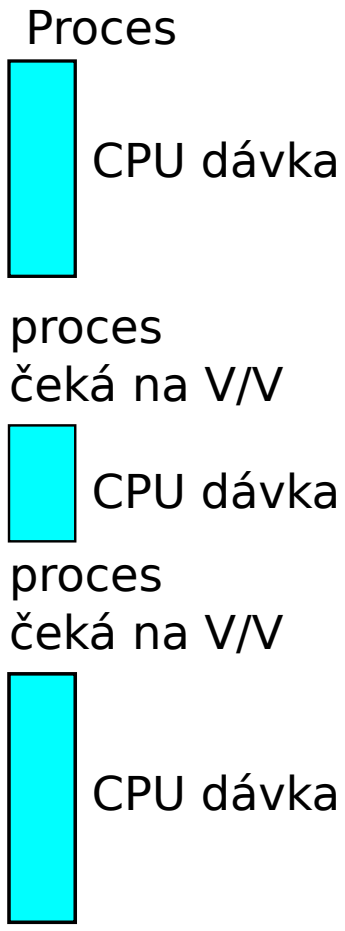
- SRT = Shortest Remaining Time (nejkratší zbývající čas)
- Preemptivní varianta SPN
- CPU dostane proces, který potřebuje nejméně času do svého ukončení
- Jestliže existuje proces, kterému zbývá k jeho dokončení čas kratší, než je čas zbývající do skončení procesu běžícího, dojde k preempci
- Může existovat více procesů se stejným zbývajícím časem, a pak je nutno použít „arbitrážní pravidlo“, např. vybrat první z fronty



- Průměrné čekání $T_{Avg} = \frac{0+7+0+9+0}{5} = 3.2$

Jak nejlépe využít procesor

- Maximálního využití CPU se dosáhne uplatněním multiprogramování
- Jak ?
- Běh procesu = cykly alternujících dávek
  - CPU dávka
  - I/O dávka
- CPU dávka se může v čase překrývat s I/O dávkami dalších procesů



Odhad délky běhu

- Délka příští dávky CPU skutečného procesu je známa jen ve velmi speciálních případech
  - Délka dávky se odhaduje na základě nedávné historie procesu
  - Nejčastěji se používá tzv. exponenciální průměrování
- Exponenciální průměrování
  - t_n skutečná změřená délka n-té dávky CPU
  - τ_{n+1} odhad délky příští dávky CPU
  - $\alpha, 0 \leq \alpha \leq 1$ parametr vlivu historie
  - $\tau_{n+1} = \alpha \cdot t_n + (1-\alpha)\tau_n$
  - Příklad:
    - $\alpha = 0.5$
    - $\tau_{n+1} = 0.5 \cdot t_n + 0.5 \cdot \tau_n = 0.5 \cdot (t_n + \tau_n)$
    - τ_0 se volí jako průměrná délka CPU dávky v systému nebo se odvodí z typu nejčastějších programů

Prioritní plánování

- Každému procesu je přiřazeno prioritní číslo
  - Prioritní číslo – preference procesu při výběru procesu, kterému má být přiřazena CPU
  - CPU se přiděluje procesu s nejvyšší prioritou
  - Nejvyšší prioritě obvykle odpovídá (obvykle) nejnižší prioritní číslo
    - Ve Windows je to obráceně
- Existují opět dvě varianty:
  - Nepreemptivní
    - Jakmile se vybranému procesu procesor předá, procesor mu nebude odňat, dokud se jeho CPU dávka nedokončí
  - Preemptivní
    - Jakmile se ve frontě připravených objeví proces s prioritou vyšší, než je priorita právě běžícího procesu, nový proces předběhne právě běžící proces a odejme mu procesor
- SPN i SRT jsou vlastně případy prioritního plánování
  - Prioritou je predikovaná délka příští CPU dávky
  - SPN je nepreemptivní prioritní plánování
  - SRT je preemptivní prioritní plánování

Prioritní plánování – problémy

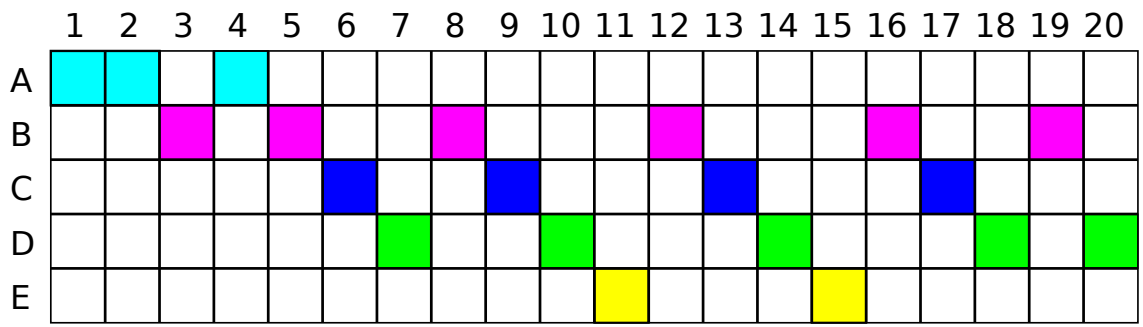
- Problém stárnutí (starvation):
  - Procesy s nízkou prioritou nikdy nepoběží; nikdy na ně nepřijde řada
    - Údajně: Když po řadě let vypínali v roce 1973 na M.I.T. svůj IBM 7094 (jeden z největších strojů své doby), našli proces s nízkou prioritou, který čekal od roku 1967.
- Řešení problému stárnutí: zrání procesů (aging)
  - Je nutno dovolit, aby se procesu zvyšovala priorita na základě jeho historie a doby setrvávání ve frontě připravených
    - Během čekání na procesor se priorita procesu zvyšuje

Cyklické plánování

- Cyklická obsluha (Round-robin) – RR
- Z principu preemptivní plánování
- Každý proces dostává CPU periodicky na malý časový úsek, tzv. časové kvantum, délky q (desítky ms)
- V „čistém“ RR se uvažuje shodná priorita všech procesů
- Po vyčerpání kvanta je běžícímu procesu odňato CPU ve prospěch nejstaršího procesu ve frontě připravených a dosud běžící proces se zařazuje na konec této fronty
- Je-li ve frontě připravených procesů n procesů, pak každý proces získává $\frac{1}{n}$ doby CPU
- Žádný proces nedostane 2 kvanta za sebou (samozřejmě pokud není jediný připravený)
- Žádný proces nečeká na začátek přidělení CPU déle než $q \cdot (n-1)$

Cyklické plánování

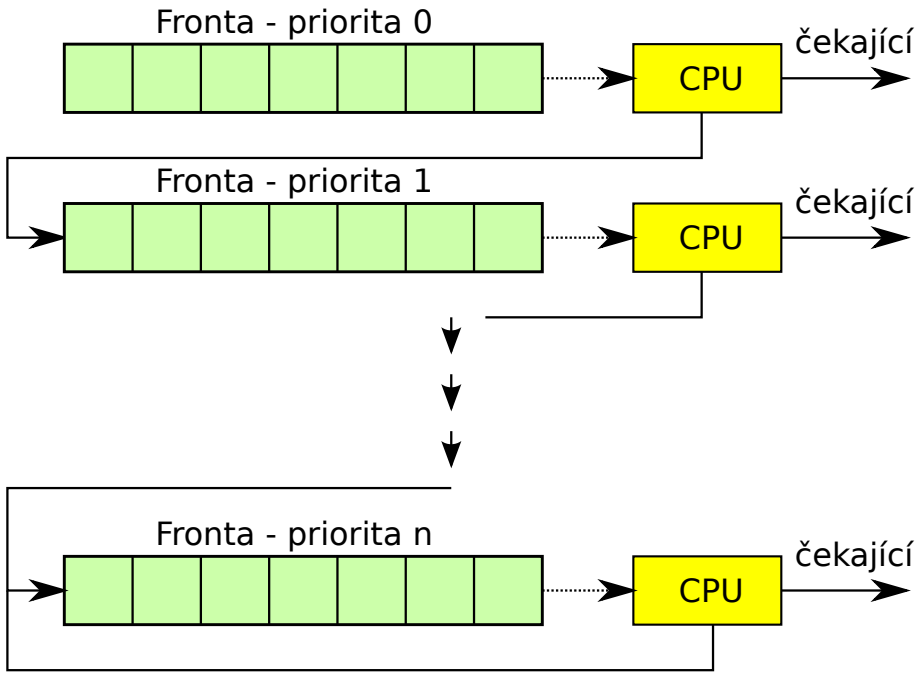
- Efektivita silně závisí na velikosti kvanta
- Veliké kvantum – blíží se chování FCFS
  - Procesy dokončí svoji CPU dávku dříve, než jim vyprší kvantum.
- Malé kvantum – časté přepínání kontextu
  - značná režie
- Dosahuje se průměrné doby obrátky delší oproti plánování SRT
  - Průměrná doba obrátky se může zlepšit, pokud většina procesů se době **q** ukončí
  - Empirické pravidlo pro stanovení **q**: cca 80% procesů by nemělo vyčerpat kvantum
- Výrazně lepší je čas odezvy



Zpětnovazební plánování

- Základní problém:
  - Neznáme předem časy, které budou procesy potřebovat
- Východisko:
  - Penalizace procesů, které běžely dlouho
- Řešení:
  - Dojde-li k preempci přečerpáním časového kvanta, procesu se snižuje priorita
  - Implementace pomocí víceúrovňových front
    - pro každou prioritu jedna
    - Nad každou frontou samostatně běží algoritmus určitého typu plánování, obvykle RR s různými kvanty a FCFS pro frontu s nejnižší prioritou

Víceúrovňové zpětnovazební fronty



- Proces opouštějící procesor kvůli vyčerpání časového kvanta je přeřazen do fronty s nižší prioritou
- Fronty s nižší prioritou mohou mít delší kvanta
- Problém stárnutí ve frontě s nejnižší prioritou
  - Řeší se pomocí zrání (aging) – v jistých časových intervalech (≈ 10 s) se zvyšuje procesům priorita přemístěním do „vyšších“ front

O(1) plánovač – Linux 2.6.22

- O(1) – rychlost plánovače nezávisí na počtu běžících procesů – je rychlý a deterministický
- Dvě sady víceúrovňových front
  - Na začátku první sada obsahuje připravené procesy, druhá je prázdná
  - Při vyčerpání časového kvanta je proces přeřazen do druhé sady front do nové úrovně
  - Vzbuzené procesy jsou zařazovány podle toho, zda ještě nevyužily celé svoje časové kvantum do aktivní sady front, nebo do druhé sady front
  - Pokud je první sada prázdná, dojde k prohození první a druhé sady front procesů
- Heuristika pro odhad interaktivních procesů a jejich udržování na nejvyšších prioritách s odpovídajícími časovými kvanty

Zcela férový plánovač

- Linux od verze 2.6.23 (Completely Fair Scheduler)
- Nepoužívá fronty, ale jednu strukturu, která udržuje všechny procesy uspořádané podle délky již spotřebovaného času a délky čekání
  - kritérium = spotřebovaný_čas - férový_čekací_čas
  - férový_čekací_čas je reálný čas dělený počtem čekajících procesů na jeden procesor
  - ideálně všechny procesy mají kritérium 0
- Pro rychlou implementaci se používá vyvážený binární červeno-černý strom, zaručující složitost úměrnou log(n) počtu připravených procesů
- Nepotřebuje složité heuristiky pro detekci interaktivních procesů
- Jediný parametr je časové kvantum:
  - pro uživatelské PC se volí menší
  - pro serverové počítače větší kvanta omezují režii s přepínáním procesů a tím zvyšuje propustnost serveru
- Žádný proces nemůže zestárnout, všechny procesy mají stejné podmínky

Kvíz

Nejhorší vlastnost plánovače je, že v něm může proces stárnout. Který z následujících plánovačů má problém se stárnutím procesů?

A - Víceúrovňové zpětnovazební fronty - Multilevel feedback queue

B - Cyklický plánovač - Round robin

C - O(1) plánovač

D - Zcela férový plánovač - Completely Fair Scheduler

Plánování v multiprocesorech

Přiřazování procesů (vláken) procesorům:

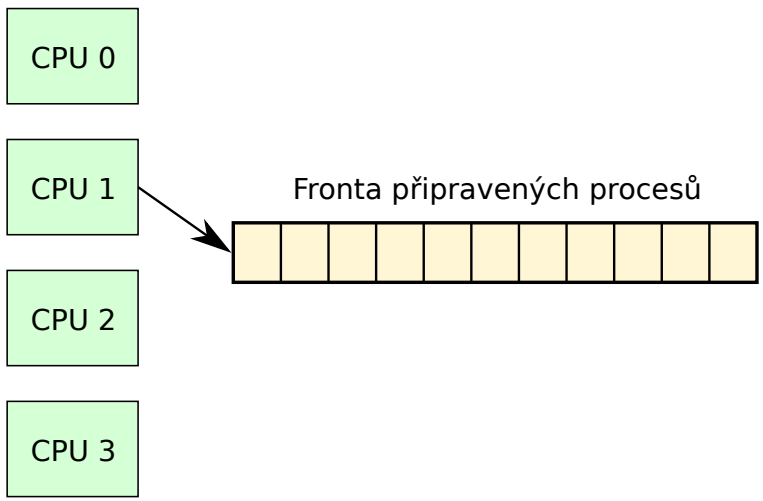
- Architektura „master/slave“
  - Klíčové funkce jádra běží vždy na jednom konkrétním procesoru
  - Master odpovídá za plánování
  - Slave žádá o služby mastera
  - Nevýhoda: dedikace
    - Přetížený master se stává úzkým místem systému
- Symetrický multiprocesing (SMP)
  - Všechny procesory jsou si navzájem rovny
  - Funkce jádra mohou běžet na kterémkoliv procesoru
  - SMP vyžaduje podporu vláken v jádře
  - Proces musí být dělen na vlákna, aby SMP byl účinný
- Aplikace je sada vláken pracujících paralelně do společného adresního prostoru
- Vlákno běží nezávisle na ostatních vláknech svého procesu
- Vlákna běžící na různých procesorech dramaticky zvyšují účinnost systému

používá většina OS: Windows, Linux, Mac OS X, Solaris, BSD4.4

SMP

Dvě řešení SMP:

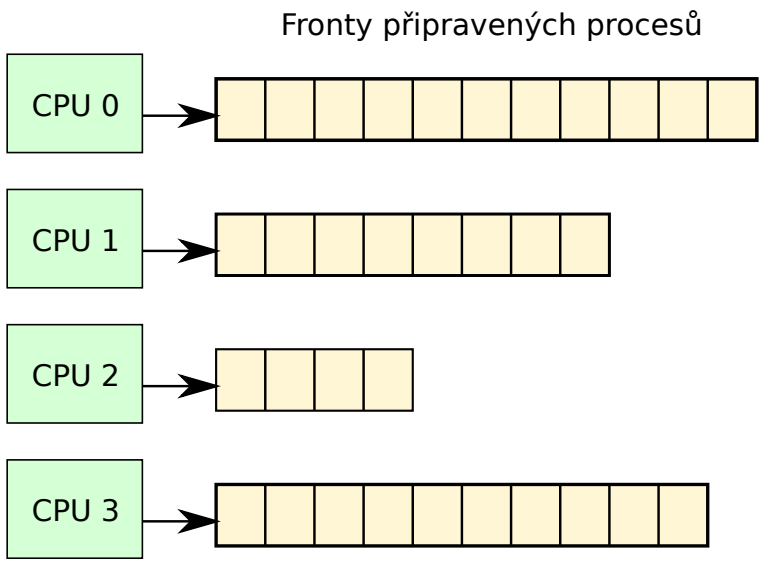
- Jedna společná fronta pro všechny procesory
  - Fronta může být víceúrovňová dle priorit
  - Problémy:
    - Jedna centrální fronta připravených sledů vyžaduje používání vzájemného vylučování v jádře
    - Kritické místo v okamžiku, kdy si hledá práci více procesorů
    - Předběhnutá (přerušená) vlákna nebudou nutně pokračovat na stejném procesoru – nelze proto plně využívat cache paměti procesorů



SMP

Druhé řešení SMP:

- Každý procesor má svojí frontu a občasná migrace vláken mezi procesory má za úkol udržovat fronty přibližně stejně dlouhé
  - Každý procesor si sám vyhledává příští vlákno
  - Přesněji: instance plánovače běžící na procesoru si je sama vyhledává
  - Problémy – některé fronty jsou kratší:
    - Heuristická pravidla, kdy frontu změnit



SMP optimalizace

- Používají se různá (heuristická) pravidla (i při globální frontě):
  - Afinita vlákna k CPU – použij procesor, kde vlákno již běželo (možná, že v cache CPU budou ještě údaje z minulého běhu)
  - Afinita vlákna k CPU při globální frontě – neber první proces z fronty, ale prozkoumej více procesů na začátku fronty a hledej proces, který běžel na daném procesoru
  - Použij nejméně využívaný procesor
- Mnohdy značně složité
  - při malém počtu procesorů (≤ 4) může přílišná snaha o optimalizaci plánování vést až k poklesu výkonu systému, výběr se dělá při každém rozhodování, kdo poběží
    - Tedy aspoň v tom smyslu, že výkon systému neporoste lineárně s počtem procesorů
  - při velkém počtu procesorů dojde naopak k „nasycení“, neboť plánovač se musí věnovat rozhodování velmi často (končí CPU dávky na mnoha procesorech)
    - režie tak neúměrně roste