Funkce operačního systému, struktura a rozhraní operačního systému, mikrojádro.

Z Na státnice zvesela!

Obsah

- 1 Funkce operačního systému
- 2 Struktura operačního systému
- 3 Rozhraní operačního systému
  - 3.1 uživatelské
  - 3.2 programové
- 4 Mikrojádro

Funkce operačního systému

- sada programů pro efektivni využití počítače
- hostuje aplikační software
- **správce zdrojů**
  - OS spravedlivě řídí přidělování zdrojů jako je paměť, čas procesoru, přístup k V/V zařízení
  - operační systém vlastní jednotlivé systémové zdroje - přiděluje a odebírá je jednotlivým procesům.
- **rozšířený stroj**
  - OS skrývá detaily ovládání jednotlivých zařízení (transparentnost), mezivrstva mezi hardware a software
  - definuje standardní rozhraní pro volání systémových služeb
  - programátor se může věnovat vlastní úloze a nemusí znovu programovat I/O operace
  - program může díky izolaci od konkretních zařízení pracovat i se zařízeními, o kterých jeho autor v době vytváření programu neměl ani ponětí (program se o ovládání I/O nestará).

Struktura operačního systému

- **monolitcké systémy** - hlavní program, obslužné procedury, podpůrné procedury
- **vrstvené systémy** - hierarchie vrstev, nejnižší je holý počítač, nejvyšší je aplikační program
- **funkční hierarchie** - někdy je problém rozdělit do vrstev podle úrovně abstrakce, proto dělení do vrstev podle funkčnosti
- **klient-server** - obsahuje mikrojádro, které poskytuje pouze základní funkce, většinu práce dělají servery, které jsou oddělené od jádra
- **objektově orientovaná struktura** - jádro spravuje řadu objektů (zastupují soubory, HW zařízení, ...), mezi objekty jsou tzv. capability = odkaz na objekt + množina práv definujících operace

Rozhraní operačního systému

uživatelské

- soubor aplikačních programů jako např. interpret příkazů (shell)
- příkazy pro práci se soubory
- správa systému
- ovládání přes příkazový řádek nebo GUI

programové

- soubor systémových volání, které poskytuje jádro OS

Instrukce lze provádět v **privilegovaném** (na žádost programu, zpracování výjimky, zpracování přerušení) nebo **neprivilegovaném** (vykonání aplikačních programů) režimu.

Mikrojádro

- vrstva nad holým strojem, která obsahuje minimální množinu abstrakcí, tak aby ostatní funkce OS mohly být implementovány nad ním.
- mikrojádro musí být vykonáváno v privilegovaném režimu a zajišťuje přerušení, nízkoúrovňový V/V, správu vláken, správa paměti (JavaOS), meziprocesovou komunikaci
- ostatní funkce - soubory, adresáře, síťové služby - jsou programy vykonávané v uživatelském režimu

- Stránka byla naposledy editována 9. 8. 2010 v 16:35.

Případová studie OS Linux.

Z Na státnice zvesela!

Charakteristika Linuxu

- **OS Unixového typu** - filozofie, procesy, uživatelé, souborový systém, základní programy a další věci jsou shodné s Unixovými standardy.
- **Čistě 32/64 bitový OS** - Linux od počátku byl psán jako 32-bitový OS a dnes podporuje řadu 64-bitových architektur
- **Víceúlohový OS** - jeden člověk může mít spuštěno několik programů současně
- **Víceuživatelský OS** - více lidí může současně pracovat na jednom fyzickém počítači. OS uživateli vytváří virtuální prostředí tvářící se, jako by měl počítač sám pro sebe: nikdo nebude bez jeho povolení číst jeho soubory, nikdo nebude zasahovat do běhu jeho programů, bude moci používat periferní jednotky počítače (tiskárny, vstupní jednotky,..) atd.
- **Víceprocesorový OS** - SMP(symetrický multi-processing) podle dané architektury - podporováno až 64 procesorů. OS zaručuje rovnoměrné využití procesorů jednotlivými procesy
- **Preemptivní OS** - žádná úloha si nemůže "přivlastnit" a zablokovat systém; systém po určité době přidělení sám odebírá úlohám procesor(y). Úloha si vůbec nemusí uvědomovat existenci střídání se o procesor.
- **Real-time OS** - kromě normálních typů procesů (-úloh) jsou podporovány i real-time procesy. Jsou jinak plánovány a mají vyšší prioritu než všechny ostatní procesy.
- **Všestranně nasaditelný** - používá se od tzv. zapouzdřených (angl. embedded) systémů (specializovaná nasazení většinou na mini HW a speciálních periferiích - řídící systémy, roboti, telefony, PDA...) přes PC, servery po clustery atd.
- **Svobodný SW** - jádro Linuxu je šířeno včetně zdrojových kódů. Kdokoliv může zdrojové kódy volně používat, upravovat a šířit - viz GNU GPL licence. Jádro i aplikační programy jsou vyvíjeny a spravovány tisíci nadšenci po celém světě komunikujícími po Internetu. Současně je ale na Linux portováno a pro něj vyvinuto mnoho komerčních programů, zpravidla za daleko nižší cenu než odpovídající verze pro komerční Unixové systémy.
- **Nejrychleji se rozvíjející OS** - za více než 15 let existence GNU/Linux vyrostl od původní verze (na i386, se souborovým systémem Minix a z programů pouze překladač C a shell) k dnešnímu stavu (jádra 2.6.x, zdrojové soubory zabírají již přes 200MB) - podpora více než 20 HW architektur, SMP, několika desítek souborových systémů a řada dalších vlastností v hlavním vývojovém stromu jádra. K tomu je nepřeberná řada jádro obalujících GNU systémových, uživatelských a dalších programů, několik X-Window manažerů,...

Charakteristiky programů pro Unix

- **Jednoduchost** - valná většina utilit pro operační systém Unix je velmi jednoduchých a v důsledku toho také malých a snadno pochopitelných (KISS - Keep It Small and Simple).

- **Úzké zaměření** - vždy je lepší vytvořit program, který provádí dobře jen jeden úkol. V systému Unix jsou v případě potřeby malé utility často kombinovány tak, aby prováděly náročnější úkoly, místo aby se programátoři snažili předvídat potřeby uživatelů za pomoci jednoho velkého programu.
- **Znovu použitelné komponenty** - je užitečné dát jádro aplikace k dispozici v podobě knihovny. Dobře dokumentované knihovny disponují jednoduchým ale flexibilním rozhraním, mohou ostatním lidem pomoci při vývoji různých variací nebo při aplikaci postupů v nových oblastech.
- **Filtry** - spoustu unixových aplikací lze využít jako filtry. To znamená, že mohou převádět vstup na výstup jiného typu.
- **Otevřené formáty souborů** - nejúspěšnější a nejoblíbenější unixové programy používají konfigurační a datové soubory, které mají podobu textových souborů.
- **Flexibilita** - nelze dopředu předvídat, jak důmyslní uživatelé budou program používat. Je proto dobré při programování zachovávat co největší flexibilitu.

- Stránka byla naposledy editována 3. 6. 2011 v 09:03.

Zavedení operačního systému.

Z Na státnice zvesela!

BIOS

- program přítomný ve vestavěné paměti HW (většinou na základní desce)
- provádí testy a nastavení HW
- vybere zaváděcí jednotku
- načte první sektor (MBR), kde je umístěn program zavaděče a provede skok na adresu jeho programu, čímž mu předá řízení

Zavaděč

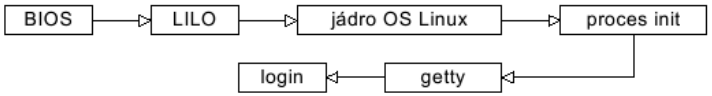
- pro Linux LILO nebo GRUB
- dává možnost zvolit startující OS
- načte jádro operačního systému do paměti a spustí ho

Jádro OS

- detekuje hardware a odpovídajícím způsobem nastaví ovladače zařízení
- připojí kořenový svazek pro čtení a provede kontrolu souborového systému
- spustí na pozadí proces init

Proces init

- proces init se konfiguruje pomocí souboru /etc/inittab
- inicializuje operační systém
- spuštěn po celou dobu běhu operačního systému a ošetřuje některé události (úklid v adresáři /tmp)
- spustí služby - Démony
- nakonec spustí program getty pro terminály a virtuální konzoli a v ní program *login*



- Stránka byla naposledy editována 13. 6. 2008 v 21:07.

Proces a jádro.

Z Na státnice zvesela!

Proces

- jedna instance vykonávaného programu
- souběžně se může vykonávat mnoho procesů, ale v jeden okamžik je čas procesoru přidělen pouze jednomu procesu

Proces v UNIXu

- proces je jednotka (entita), která vykonává programy a poskytuje prostředí pro jejich vykonávání
- adresový prostor + počítadlo instrukcí
- proces je základní jednotkou plánování (scheduling)
- **kontext procesu**
  - uživatelský adresový prostor
    - uživatelský zásobník
    - text
    - data
  - řídící informace
    - proc záznam
    - u oblast
    - zásobník jádra
  - registry
    - PSW (Program Status Word)
    - stack pointer
    - instruction pointer
    - paměťové registry

Každý proces má také tzv. **u (user) oblast** a položku v tabulce procesů, tzv. **proc záznam (též deskriptor procesu)**.

- **u oblast**
  - údaje potřebné když je proces vykonávaný
  - tabulku deskriptorů otevřených souborů
  - okamžitý adresář
  - kořenový adresář
  - často zásobník jádra procesu

- **proc záznam**
  - položka tabulky procesů (alespoň jsem to tak pochopil)
  - identifikaci PID
  - umístnění u oblasti
  - stav
  - ukazatele na vytvoření seznamů všech procesů, čekajících procesů, ...
  - událost, na kterou proces čeká
  - informace pro plánování
  - pole neošetřených signálů
  - informace pro správu paměti
  - propojení na PID v rozptýlené (hash) tabulce

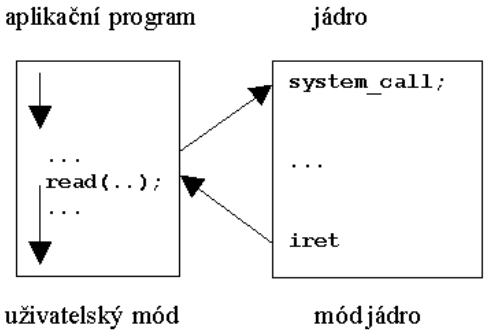
- informace pro hierarchii procesů

virtuální adresový prostor procesu se skládá z adresového prostoru procesu (uživatelského) a systémového prostoru (jádra). Proces v uživatelském módu má přístup ke svému adresovému prostoru. K systémovému prostoru má přístup jen voláním system_call(). Jádro oproti tomu přístup k adresovému prostoru procesů má.

- Služba jádra může být obecně volána více programy najednou, proto musí být jádro **reentrantní**. Každý proces proto má svůj zásobník jádra, často v adresovém prostoru procesu – chráněný, spravovaný jádrem.

Jádro

- speciální program zavedený do hlavní paměti při startu systému přímo vykonávaný HW



- **vykonávaný program** – proces = posloupnost instrukcí aplikačního (uživatelského) programu a jádra
- Služba jádra může být obecně volána více programy najednou, proto musí být jádro **reentrantní**. Každý proces proto má svůj zásobník jádra, často v adresovém prostoru procesu – chráněný, spravovaný jádrem.

jádro

- vykonává služby
- zpracovává výjimky (pokus dělit nulou, přetečení uživatelského zásobníku, ...)
- zpracovává přerušení od periferních zařízení
- vykonává systémové procesy (správa paměti, přepočítávaní priorit procesů)

jádro pracuje

- v kontextu procesu (systémová volání, výjimky)
- v systémovém kontextu (přerušení, systémové procesy)

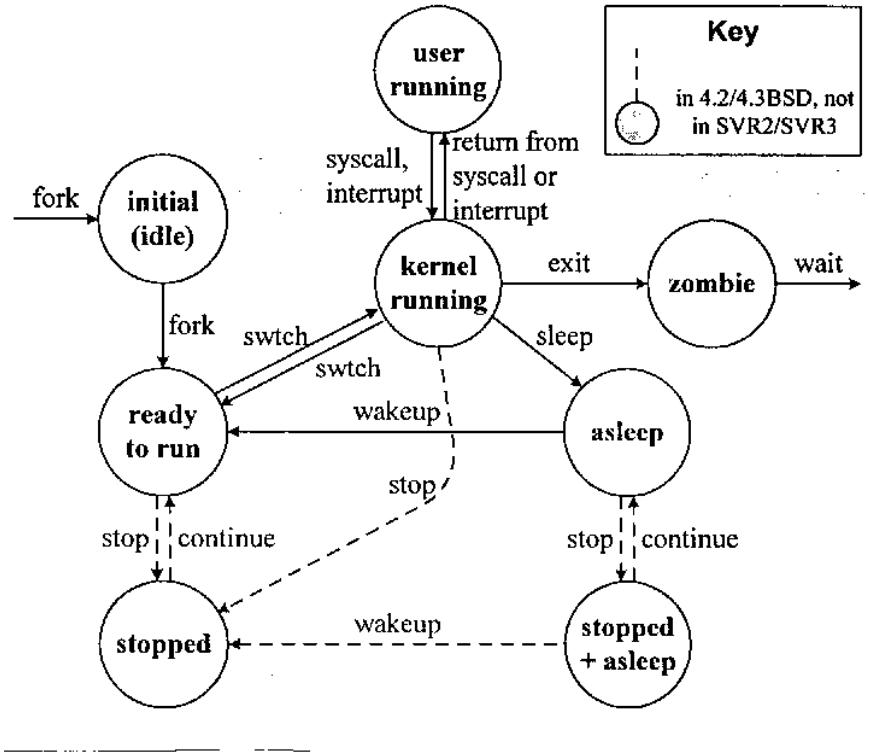
Citováno z „http://wiki.zvesela.cz/index.php/Proces_a_j%C3%A1dro."

- Stránka byla naposledy editována 5. 8. 2010 v 08:36.

Stavy procesu + stavy vlákna, oprávnění uživatele.

Z Na státnice zvesela!

Stavy procesu



Process states and state transitions.

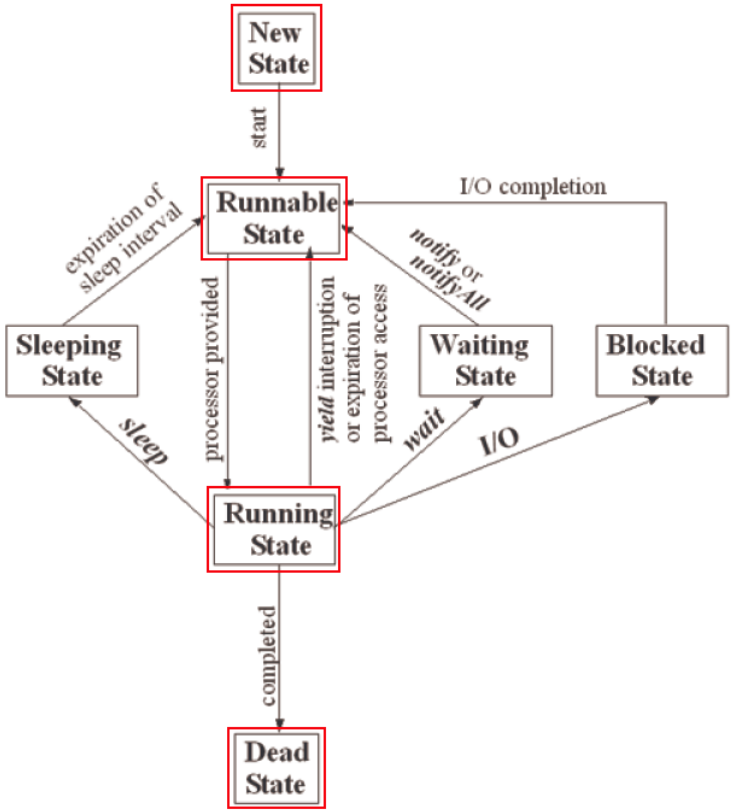
- **Počáteční** (initial/idle) – fork začal vytváření procesu.
- **Připraven na vykonání** (ready to run) – fork dokončil vytváření procesu, proces čeká až bude naplánován na přidělení procesoru.
- **Běžící v jádře** (kernel running) – byl naplánován, vykoná se přepnutí kontextu, procedura jádra swtch() uloží HW kontext do registrů.
- **Běžící uživatelsky** (user running) – proces vykonává svůj programový kód. Může přejít do stavu běžící v jádře v důsledku volání služby jádra nebo přerušení, po skončení obsluhy se vrátí.
- **Spící** (asleep) – při vykonávání systémového volání se může stát, že je nutno čekat na nějakou událost nebo

prostředek, proces (v jádře) proto zavolá proceduru sleep(). Když událost nastane, jádro vzbudí proces, proces se stane připraven na vykonání a po naplánování pokračuje obsluha systémového volání ve stavu běžící v jádře.

Připraven na vykonání se může stát také, je-li běžící a uplyne mu přidělené časové kvantum - vykoná se preempce běžícího procesu a to ve stavu běžící uživatelsky nebo při návratu do něj. Jádro je nepreemptivní. Přerušení se může vyskytnout i ve stavu běžící v jádře, kdy po skončení obsluhy přerušení proces pokračuje ve stavu běžící v jádře.

- **Mátoha** (zombie) - proces končí voláním exit() anebo v důsledku signálu, dokud rodič nevykoná wait().
- **Zastaven** (stopped) - do něj přejde proces, je-li běžící nebo spící (+ asleep), po stop signálech:
  - SIGSTOP zastav proces
  - SIGTSTP CTRL-Z
  - SIGTTIN tty čtení procesu v pozadí
  - SIGTTOU tty psaní procesu v pozadí
  - SIGCONT převede proces do stavu připraven na vykonání nebo do stavu spící.

Stavy vlákna



Oprávnění uživatele

- každý uživatel je v systému identifikován číslem - identifikátor uživatele (user identifier) UID, obdobně identifikátor skupiny (group identifier) GID
- identifikátory ovlivňují vlastnictví vytvářených souborů, používají se na kontrolu přístupových práv a kontrolu zasílaní signálů jiným procesům
- procesy dědí oprávnění
- privilegovaný uživatel superuser UID 0, GID 1
- každý proces má
  - reálný UID (real UID) - identifikuje reálného uživatele a ovlivňuje právo posílat signály
  - efektivní UID (effective UID) - ovlivňují vlastnictví souborů a přístup k souborům
  - uložený nastavovací UID (saved set UID)
- proces změní efektivní UID
  - vykoná-li exec programu s nastaveným bitem SUID
  - systémovým voláním setuid
- bit SUID, nastavení UID (set UID), je jeden z bitů "přístupových" práv k souboru
  - je-li nastaven, efektivní UID a uložený nastavený, UID se nastaví na ID vlastníka souboru
- setuid(uid)
  - je-li okamžitý efektivní UID procesu privilegovaný uživatel, potom reálný UID, efektivní UID a uložený nastavovací UID se nastaví na uid
  - jinak setuid(uid)nastaví efektivní UID na hodnotu uid, je-li uid rovno reálnému UID nebo uloženému nastavovacímu UID, reálný UID a uložený nastavovací UID se nezmění

- přístup k souborům je rozdělen do tří skupin:
  - **read (r)**
  - **write (w)**
  - **execute (x)**
- každý soubor obsahuje tři trojice těchto bitů a to pro:
  - **owner** - autor, který soubor vytvořil
  - **group** - skupina, do které autor patří
  - **others** - všichni ostatní uživatelé

- Stránka byla naposledy editována 6. 8. 2010 v 06:59.

Proces, thread a fibre – implementace a využití.

Z Na státnice zvesela!

Proces

- může mít jedno nebo více vláken, vykonávaných v tomtéž adresovém prostoru a sdílejících prostředky procesu, např. soubory

Lehký proces (light-weight process)

- lehký proces je uživatelské vlákno podporované jádrovým vláknem
- proces může mít jeden nebo více lehkých procesů, každý podporovaný zvláštním jádrovým vláknem
- je nezávisle plánován OS, proto může běžet paralelně
- sdílí adresový prostor a většinu ostatních prostředků procesu
- obsahuje pouze minimální HW kontext a statistické informace nutné pro plánovač

Vlákno (thread)

- je relativně nezávislá část aplikace vykonávaná sekvenčně, která má jeden tok řízení a může být plánována operačním systémem, existuje v procesu a může používat jeho prostředky
- každé vlákno má vlastní zásobník, počítadlo instrukcí a HW kontext
- efektivní řešení v porovnání s procesy, vyžaduje však synchronizaci na úrovni vláken
- **jádrová vlákna** - vytvářena a rušena uvnitř jádra pro určené funkce, sdílí prostor jádra a má vlastní zásobník, není sdruženo se žádným uživatelským procesem, koncepčně shodné s démony
- **uživatelská vlákna (fibers)** – implementovány jako knihovní fce (PThreads), interakce vláken nezahrnují jádro a jsou proto velice rychlé, nemůžou využít paralelizmus multiprocesorových systémů

Implementace vlákna

- **Linux** - poskytuje platformově závislé systémové volání __clone() a má 4 parametry:
  - **fn** - specifikuje funkci, kterou má potomek vykonat, po jejím vykonání potomek končí
  - **arg** - ukazatel na argumenty funkce fn()
  - **flags** - obsahuje signál, který se pošle rodiči po skončení potomka a příznaků klonování, které specifikují části kontextu (prostředky) sdílené potomkem a rodičem
  - **child_stack** - specifikuje ukazatel na uživatelský zásobník potomka
- **POSIX (Portable Operating System Interface) 1003.1c – PThreads (POSIX Threads)**
  - vytvoření vlákna – pthread_create()
  - inicializace vlákna – pthread_start_thread
  - ukončení vlákna – pthread_exit(stav)
  - čekání na skončení vlákna – pthread_join (thread_id, stav)
  - vzájmen vyloučení – mutex(synchronizace přístupu k prostředkům (datům)), podmínkové proměnné (synchronizace při dosažení nějakých hodnot proměnných)
  - vlákna jsou často mapována na jádrová (plánovaná OS) a umožňují tak využití více procesorů
- neposkytuje-li OS podporu vláken lze je implementovat také pomocí lehkých procesů (umí-li je OS)

Fiber (uživatelská vlákna)

- fiber má k vláknu stejný vztah, jako má vlákno k procesu
- vykonání fibre musí být v rámci vlákna ručně naplánováno, tj. fiber se explicitně vzdává procesoru
- fiber běží v kontextu vlákna
- každé vlákno může plánovat více fibers
- nejsou-li fibers mapována na jádrová vlákna, nemohou využít multiprocesoru
- přepínání mezi těmito vlákny je extrémně efektivní, protože nepoužívají žádná syst. volání a nemusí se přepínat celý kontext

- Stránka byla naposledy editována 5. 8. 2010 v 15:41.

Proces, thread a fibre – implementace a využití. - Na státnice zvesela! http://wiki.zvesela.cz/index.php/Proces,_thread_a_fibre_–_impleme...

Výpočet v módu jádro, systémová volání, výjimky, přerušení, signály.

Z Na státnice zvesela!

Výpočet v módu jádro

- v důsledku událostí - přerušení (od zařízení asynchronně), výjimky, softwarové přerušení
- řízení se předá na proceduru pro ošetření odpovídající události
- část stavu přerušeného procesu, potřebná pro obnovení jeho vykonávání (počítadlo instrukcí, PSW) se uloží po skončení obsloužení události do zásobníku jádra přerušeného procesu

Popis kernel módu (ENG) (http://www.linfo.org/kernel_mode.html)

Systémové volání

- ve standardní knihovně jazyka C je pro každé systémové volání obálková procedura
- řízení se předá softwarovým přerušením proceduře jádra, která se nazývá syscall(), system_call (uloží obsah registrů (HW kontext))
- zavolá odpovídající funkci
- ukončí se voláním ret_from_sys_call())
- požadovaná služba je identifikována parametrem procedury, který se nazývá číslo systémového volání

Systémové volání (ENG) (http://www.linfo.org/system_call.html) Obrázek systémového volání (http://www.linux.it/~rubini/docs/ksys/ksys.html)

Výjimky

- jsou synchronní s procesem (vznikají v důsledku událostí způsobených vykonáváním procesu)
- procedury pro jejich zpracování mají obdobnou strukturu jako procedura pro systémová volání system_call
- Procedura pro zpracování výjimky:
  - uloží obsah registrů
  - zpracuje výjimku
  - pošle signál procesu
  - zpracuje žádost o stránku
  - ukončí se voláním funkce ret_from_exception()

Přerušení (hardwarové)

- přerušení je obecně asynchronní vzhledem k přerušenému procesu
- zpracování přerušení nesmí způsobit čekání, přerušený proces zůstává ve stavu běžící
- při zpracování přerušení se tedy přistupuje do jeho záznamu proc
- obsloužení přerušení - uloží IRQ (Interrupt ReQuest) a obsah registrů, pošle potvrzení PIC (Programmable Interrupt Controller), vykoná obslužní proceduru přerušení, ukončí se skokem na ret_from_intr()
- přerušení mají přiřazené prioritní úrovně

Více o přerušení (http://cs.wikipedia.org/wiki/P%C5%99eru%C5%A1en%C3%AD)

Výpočet v módu jádro, systémová volání, výjimky, přerušení, signály.... http://wiki.zvesela.cz/index.php/Výpočet_v_módu_jádro,_systémov...

Signály

- umožňují oznámit procesům výskyt událostí v systému
- krátké zprávy, kde se procesům oznámí číslo signálu
- systémová volání pro signály i vnitřní implementace se u jednotlivých variant a verzí značně liší
- čísla některých signálů závisí na HW, označují se symbolickými konstantami SIG
- slouží tedy k uvědomění procesu, že nastala určitá událost, čímž jej přinutí vykonat funkci na zpracování signálu
- systémová volání umožňují programátorovi zasílat signály (funkce kill(pid,sig)) a určit jak budou použity
- jádro při zasílaní signálů rozlišuje dvě fáze - odeslání signálu(jádro zaznamená v záznamu proc (deskriptoru procesu) zasílanému procesu odeslání nového signálu), přijetí signálu (jádro přinutí proces reagovat na signál)
- pro každý signál je nastavená implicitní reakce, která se vykoná, pokud ji proces nespecifikuje jinak
  - **T** - proces je násilně ukončen
  - **A** – abort – totéž co T + vykoná nějaká akce (výpis obsahu paměti procesu)
  - **I** – ignore - signál je ignorovaný
  - **S** – stop - proces je zastaven
  - **C** – continue - byl-li proces zastaven, může pokračovat, je převeden do stavu připraven, jinak je signál ignorován
- **asynchronní signál** - stisknutí CTRL-C -> generuje se přerušení (jako u každého stisknutí klávesy) -> ovladač rozpozná, že jde o kombinaci generující signál a odešle signál SIGINT procesu v popředí, proces najde signál: při návratu do uživatelského módu po naplánování; při návratu z přerušení běžel-li
- **synchronní signál** - výjimka způsobí přechod do módu jádro, jádro vykoná její obsluhu a zašle se odpovídající signál běžícímu procesu, při návratu z obsluhy proces najde signál
- **nespolehlivé signály** - funkce pro obsluhu signálů nejsou perzistentní - při instalaci obslužní funkce pomocí signal(...) -> pokud nechceme aby se vykonala implicitní akce, musíme před každým přijmutím daného signálu instalovat obslužní funkci
- **spolehlivé signály** - perzistentní obslužné funkce signálů - při instalaci obslužní funkce pomocí sigaction(...)

Citováno z „http://wiki.zvesela.cz/index.php/V%C3%BDpo%C4%8Det_v_m%C3%B3du_j%C3%A1dro%2C_syst%C3%A9mov%C3%A1_vol%C3%A1n%C3%AD%2C_v%C3%BDjimky%2C_p%C5%99eru%C5%A1en%C3%AD%2C_sign%C3%A1ly.“

- Stránka byla naposledy editována 7. 8. 2010 v 11:08.

Plánování, plánovací třídy, inverze priority.

Z Na státnice zvesela!

Obsah

- 1 Plánovací strategie
  - 1.1 Round robin
  - 1.2 Priority scheduling
  - 1.3 Shortest process next
  - 1.4 Fair-Share scheduling
  - 1.5 Plánování použitím epoch
- 2 Plánovací třídy
- 3 Inverze priority

Plánovací strategie

- pravidla na určení kdy vykonat přepnutí procesů, a který proces vybrat
- tradičně vychází ze sdílení času
- procesy podle jejich požadavků můžeme rozdělit - interaktivní procesy, dávkové procesy, procesy reálného času
- procesům, které dlouho nečerpají své časové kvantum prioritu zvýšíme a opačně procesům, které jsou dlouho běžící penalizujeme
- **volba časového kvanta** - krátké trvání způsobuje vysokou režii, příliš dlouhé trvání způsobuje ztrátu zdání souběžného vykonávání procesů => zvolit trvání časového kvanta co nejdelší při zachování dobrého času odpovědi systému
- priority v módu jádro – nepřerušitelné, přerušitelné

Round robin

Jedná se o jednoduchou plánovací strategii s předbíháním (co znamená předbíhání?). Připravené procesy čekají ve frontě typu FIFO. Plánovač postupně vybírá procesy z této fronty a přiděluje jim časová kvanta. Po uplynutí časového kvanta (pokud ještě proces neskončil) je zařazen na konec fronty. Velikost časového kvanta je důležitý parametr a v případě potřeby je jím možné řídit prioritu procesů.

Priority scheduling

Každý proces má přidělenou prioritu. Připravené procesy se pak podle své priority řadí do prioritních tříd a CPU je přidělována procesům z nejvyšší neprázdné prioritní třídy. V této prioritní třídě jsou procesy plánovány metodou Round Robin. Nevýhodou plánovací strategie je hladovění procesů s nízkou prioritou, což se může například řešit zvýšením priority příliš dlouho čekajícího procesu.

Varianty prioritního plánování:

- Bez předbíhání
- S předbíháním

- Statická priorita
- Dynamická priorita
- Fixní časové kvantum pro všechny prioritní třídy
- Různě velká časová kvanta pro různé prioritní třídy

Shortest process next

Jedná se o modifikaci plánování Shortest Job First pro interaktivní systémy. U procesů se odhaduje potřebná doba výpočtu na základě minulého chování nebo jej odhadne zadavatel, který spouští proces. Plánovač pak vybírá proces s nejmenší odhadnutou dobou výpočtu. Problémem strategie je možnost hladovění dlouhých procesů, respektive procesů, u kterých je stále odhadována dlouhá doba výpočtu. Tato strategie je aplikovatelná především na dávkové operační systémy.

Fair-Share scheduling

V tomto prioritním plánování znamená vyšší numerická hodnota nižší prioritu. Každý proces k má nastavenu pevnou základní prioritu Bk a v každém časovém intervalu i používal proces k CPU po dobu CPUk(i). Proces má v časovém intervalu i prioritu Pk(i) a na začátku každého časového intervalu je nastavena hodnota CPUk(i) na polovinu předchozí hodnoty. Hodnota Pk(i) je pak dána jako součet základní priority a nové hodnoty CPUk(i). Plánovač vybere proces s nejnižší hodnotou Pk(i). Tato plánovací strategie, jak už tomu její název napovídá, zajišťuje poctivější sdílení procesorového času než například prioritní plánování. Odstraňuje navíc problém s hladověním procesů přímo svým návrhem a nemusí ho řešit nějakým dodatečným mechanizmem.

Plánování použitím epoch

Plánovací strategie používaná např. v operačním systému Linux. Rozděluje čas procesoru do tzv. epoch. V každé epoše má každý proces přiděleno časové kvantum vypočítané na začátku každé epochy. Toto časové kvantum nemusí proces vyčerpat najednou. Pokud se například uspí při čekání na I/O operaci a nevyčerpal své časové kvantum, bude znovu naplánován ve stejné epoše. Epocha končí, když všechny běhu schopné procesy vyčerpaly svá kvanta. Následně jsou vypočtena nová časová kvanta všech procesů (nejen běhu schopných) a začíná nová epocha. Délka časového kvanta závisí na prioritě procesu.

Plánovací třídy

plánovací třídy: priority:

- reálný čas 100 – 159
- systém 60 – 99
- sdílení času 0 – 59

vysoká hodnota = vysoká priorita

- **plánovací třída pro sdílení času**
  - dynamicky mění priority a pro procesy stejné priority používá cyklické plánování
  - pro každou prioritu je definováno časové kvantum a další parametry
- **plánovací třída pro reálný čas**
  - statické priority a pevné časové kvantum
  - omezená plánovací latence + doba odpovědi

- **Linux**
  - čas procesoru je rozdělen do epoch
  - každý proces má specifikováno časové kvantum vypočteno na jejím začátku
  - v jedné epoše proces může využívat své časové kvantum po částech
  - epocha končí, když všechny běhu schopné procesy vyčerpali svá časová kvanta

- **Skryté plánování**
  - proces reálného času mající vysokou prioritu, může vyžadovat službu jádra, kterou vykonává např. jádrové vlákno nižší priority

Inverze priority

Inverzí priorit rozumíme situaci, která nastane, pokud vlákno s vyšší prioritou požaduje přístup k systémovým zdrojům, které v danou chvíli právě exkluzivně drží vlákno s nižší prioritou. V tomto případě dojde k preempci do vlákna s vyšší prioritou, které však nemůže běžet díky zablokovanému systémovému zdroji. To je velice nepříjemná situace, zejména v RTOS. Jediným řešením je umožnit vláknu, které systémový zdroj drží co nejrychleji doběhnout a umožnit tak i jiným vláknům pokračovat v jejich činnosti. **K vyřešení této situace se používá systém inverze priorit**, který umožní vláknu s nižší prioritou zdědit prioritu kritického vlákna, rychle vykonat potřebné operace až do chvíle uvolnění požadovaného systémového zdroje a dále pak nechat pokračovat v práci kritické vlákno.

Citováno z „http://wiki.zvesela.cz/index.php/Pl%C3%A1nov%C3%A1n%C3%AD%2C_pl%C3%A1novac%C3%AD_t%C5%99%C3%ADdy%2C_inverze_priority.“

- Stránka byla naposledy editována 7. 8. 2010 v 12:13.

Uvíznutí, vyhladovění, problém korupce dat – charakteristika a způsoby řešení.

Z Na státnice zvesela!

Pěkný materiál o uváznutí procesů uvedený na studentské wiki ČVUT. (http://student.cvut.cz/cwut/index.php/Uv%C3%A1znut%C3%AD_proces%C5%AF) - bohužel zřejmě již neexistuje...

Obsah

- 1 Deadlock (uvíznutí)
- 2 Livelock (aktivní uvíznutí)
- 3 Vyhladovění (starvation)
- 4 Problém korupce dat

Deadlock (uvíznutí)

Deadlock je odborný výraz pro situaci, kdy úspěšné dokončení nějaké akce je podmíněno předchozím dokončením jiné akce, přičemž tato jiná akce může být dokončena až po dokončení původní akce. Vzniká paradox, často označovaný jako 'Co bylo dříve? Slepice nebo vejce?'.

V počítači se jedná o zablokování procesů (případně vláken) způsobené čekáním na synchronizačních primitivech. Obvykle k němu dochází v důsledku chyby při jejich programování. Pokud uváznutí nastane, řeší se například zrušením transakce (*rollback*) nebo násilným ukončením procesů. Některé systémy spoléhají na to, že deadlock nastává zřídka a uživatel si pomůže sám.

Deadlock může nastat pouze v systémech, ve kterých jsou splněny následující čtyři **nutné podmínky** pro vznik deadlocku:

1. podmínka *vzájemného vyloučení*: zdroj (prostředek (angl. resource)) může být vlastněn pouze jedním procesem
2. podmínka *drž a čekej*: proces který již nějaký prostředek vlastní smí požadovat další prostředek
3. podmínka *neodebratelnosti*: pouze proces držící zdroj ho může uvolnit
4. podmínka *kruhového (cyklického) čekání*: dva či více procesů tvoří uzavřený řetěz, kdy každý čeká na zdroj držený jiným procesem

Řešení (podle ZOS)

- *Ignorace* - pštrosí algoritmus = předstíráme, že se nic neděje
- *Detekce a zotavení* - nesnažíme se zabránit, ale detekovat (např pomocí precedenčních grafů), pak provedeme rollback nebo zabití jednoho z procesů
- *Dynamické zabránění* - systém rozhodne, zda je přiřazení zdrojů bezpečné, pokud ano, zdroj přiřadí, jinak pozastaví žádající proces; Bankéřův algoritmus
- *Prevence* - uděláme vše proto, aby k deadlocku nedošlo = vyřešíme Coffmanovy podmínky
  - *vzájemné vyloučení* - před zdroj dáme frontu, neboli spool (např. u tiskárny)
  - *drž a čekej* - proces musí zabrat všechny zdroje najednou (použito např. u databází)

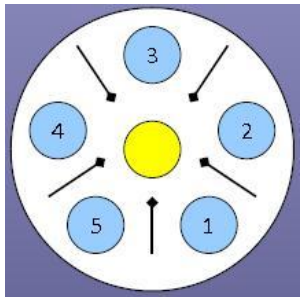
- *neodebratelnost* - to je problém, odebrání způsobí chaos
- *kruhové čekání* - přidělování zdrojů ve předem určeném pořadí

[Zdroj: ZOS přednášky, Pešička]

Livelock (aktivní uvíznutí)

- Procesy stále mění svůj stav v závislosti na druhém procesu, ale žádný nepostupuje vpřed ve svém výpočtu.
- Stejná situace, jako když jdou dva lidi proti sobě, chtějí být ohleduplní a snaží se vyhnout, přičemž se pořád vyhýbají stejným směrem, takže si neustále brání.

- Př. Večeřící filozofové
  - 5 filozofů žije pohromadě, čas tráví přemýšlením
  - pokud má filozof hlad, jde se do jídelny najíst
  - v jídelně je prostřeno pro 5 osob, na stole mísa špaget
  - k jídlu je potřeba použít 2 vidličky...
  - ... na stole je ovšem pouze 5 vidliček!
  - 1 vidlička mezi dvěma talíři
  - Algoritmus najezení, při kterém ale dojde právě k livelocku
    - všichni najednou zvednou svoji levou vydličku
    - pokud nemohu vzít druhou vidličku, položím tu první
    - všichni zvednou levou, podívají se doprava, položí levou
    - filosofové pracují ale nenají se – livelock + vyhladovění



Vyhladovění (starvation)

Nastává pokud je procesu neustále odmítnuto poskytnout požadovaný zdroj, bez kterého nemůže dokončit úkol. Večeřící filozofové. Podobné livelocku. Oproti deadlocku, proces mění své stavy, kdežto při deadlocku pasivně čeká.

Problém korupce dat

- korupce dat může nastat, pokud správně neošetříme kritickou sekci a dva procesy/vlákna paralelně zapíšou do sdílených dat
- ošetření např. mutexem

Citováno z „http://wiki.zvesela.cz/index.php/Uv%C3%ADznut%C3%AD%2C_vyhladov%C4%9Bn%C3%AD%2C_probl%C3%A9m_korupce_dat_%E2%80%93_charakteristika_a_zp%C5%AFsoby_%C5%99e%C5%A1en%C3%AD.“

- Stránka byla naposledy editována 12. 8. 2010 v 18:20.

Základní a strukturované formy interakce procesů a vláken.

Z Na státnice zvesela!

Primitivní formy interakce vláken

- **Synchronizace**
  - Zajištění návaznosti činností mezi vlákny
- **Semafor**
  - Základní prostředek pro synchronizaci vláken
  - Obsahuje frontu, ve které čekají vlákna, čítač a dvě funkce, které volají vlákna před vstupem a po výstupu z kritické sekce – P() (Proberen – čekání na semaforu) a V() (Verhogen – signál na semaforu)
  - Při synchronizaci se z vlákna A volá P() a z vlákna B se volá V()
- **Bariéra**
  - Prostředek pro synchronizaci více vláken v jeden okamžik
  - Typicky se používá, pokud více vláken provádí výpočet a pro další pokračování výpočtu je nutné dokončení dílčí části u všech vláken. Vlákna, která skončí dílčí výpočet čekají na bariéře, dokud výpočet neukončení poslední vlákno, pak mohou opět všechna pokračovat v činnosti
  - Zpravidla obsahuje čítač určující, kolik vláken se má ještě na bariéře zastavit. Dokud nedorazí všechna vlákna, všechny příchozí se uspí. Jakmile dorazí poslední vlákno, všechny vlákna čekající na bariéře se probudí
- **Sdílení dat**
  - Procesy využívají ke komunikaci datové struktury umístěné ve sdílené paměti.
  - Vlákna se navzájem nemusí znát, musí však být vyloučena současná změna dat více vlákny najednou (mutual exclusion).
  - Části programu, ve kterých může dojít ke konfliktním přístupům k datům se označují jako kritické sekce.
  - **Binární semafor**
    - Stačí pro vzájemné vyloučení přístupu k datům.
    - Zaručí, že k datům může najednou přistupovat pouze jedno vlákno
  - **Zámek**
    - Funguje podobně jako binární semafor, ale místo fronty uspaných vláken používá aktivní čekací smyčku
    - Použití zámků či semaforů může vést k zablokování aplikace.
    - Zablokování lze zabránit očíslováním zdrojů a zamykáním v určeném pořadí
- **Zasílání zpráv**
  - Na rozdíl od předchozích forem komunikace je použitelná i pro systémy s distribuovanou pamětí
  - Je třeba realizovat vyrovnávací paměť pro zprávy (fronta zpráv, send, receive)
  - Podle charakteru send a receive lze komunikaci rozdělit na:
    - Asynchronní – blokující je pouze receive
    - Synchronní – blokující je send i receive
  - Podle adresování zpráv lze komunikaci rozdělit na:
    - Symetrické – Zpráva obsahuje jak adresu příjemce tak i odesílatele
    - Asymetrické – Zpráva obsahuje jek adresu příjemce
    - Nepřímé – Zpráva obsahuje pouze adresu vyrovnávací paměti či komunikačního

kanálu, tj. odesílatel a příjemce se navzájem vůbec nemusí znát

Strukturované formy interakce vláken

- Oproti primitivním formám komunikace poskytují větší bezpečnost programování a navíc bývají často přímo implementovány v programovacím jazyce
- **Monitor**
  - Objekt poskytující vláknům služby pro konkurenční prostředí
  - Typový monitor – objektový typ (class), podle kterého se vytváří instance
  - Model výpočtu pomocí monitoru
    - Vlákna se mezi sebou vůbec neznají, každé zná monitor
    - Veškeré interakce vláken probíhají nepřímo prostřednictvím monitoru
  - Implementace monitoru obsahuje frontu, zámek, sdílená data procesů a služby monitoru (metody)
  - Vlastnosti:
    - Volání metody (služby) musí být atomické (nejde přerušit z venku, musí být zajištěna konzistence dat)
    - Implementace atomicity vyžaduje zámek
    - Někdy nelze službu poskytnout uspat vlákno (blokující volání služby), nebo vrátit error code
    - Pro blokující volání mývají monitory podporu v podobě funkcí wait() (blokující) a notify(), notifyAll() (neblokující)
    - Je potřeba fronta čekajících vláken
    - Ve wait() je nutno odemknout zámek, za wait() zase zamknout (v monitoru smí být aktivní pouze jedno vlákno)
    - V Javě je jen 1 fronta vláken, ale ta mohou čekat z různých důvodů – volat notifyAll() a vlákna opět otestují možnost běhu
    - V POSIX vláknech je fronta implementována podmínkovou proměnnou
- **Rendezvous**
  - Používaná forma je synchronní (vlákna se při rendezvous synchronizují) a asymetrická (1 vlákno v roli klient, druhé v roli server – akceptuje rendezvous, klient musí znát (mít odkaz na) server, naopak ne).
  - V každém vlákně se nachází jeho výkonný kód, v serveru se navíc nachází společný kód vláken, který se provede po zavolání entry (volá klient) a jeho akceptaci (provádí server)

Meziprocesová komunikace

- účelem je přenos údajů, sdílení dat, oznámení vzniku událostí, sdílení prostředků, sledování a řízení běhu procesu, např. při ladění programů
- **signály** - umožňují oznámit procesům asynchronní události
- **roury (pipes)** - umožňuje zapisovat data na začátek roury a číst je na konci
  - **nepojmenované roury** - vytvářejí se systémovým voláním pipe(), které vrátí dva deskriptory jeden pro čtení a druhý pro zápis
  - **pojmenované roury** - jsou perzistentní, existují jako soubory i když je nepoužívají žádné procesy
- **sledovaní procesů** - systémové volání ptrace - umožňuje sledovat a řídit běh procesu pid
- **Semafory**

```
struct sembuf {
       unsigned short sem_num;
       short sem_op;
       short sem_flg;
}
```


  - sem_op < 0 - větší nebo rovná abs. hodnotě sem_op, abs. hodnota sem_op se odečte od hodnoty semaforu, je-li menší, proces je blokován (spící) dokud není zvýšena hodnota semaforu
  - sem_op > 0 - zvýší se hodnota semaforu o hodnotu sem_op a vzbudí se procesy čekající na její zvýšení
  - sem_op = 0 - proces je blokován dokud hodnota semaforu není 0
- **Fronty zpráv**
  - zpráva vytvořená procesem je zaslána do fronty zpráv dokud ji jiný proces přečte
  - zpráva obsahuje 32 bitový typ zprávy a data zprávy
  - typ zprávy umožňuje selektivně vybírat zprávy z fronty
  - zprávy jsou ve frontě v pořadí jejich příchodu
- **Sdílená paměť**
  - sdílená paměť je oblast paměti, která je sdílená více procesy
  - proces sdílenou oblast pamětí vytvoří, procesy potom připojí oblast na virtuální adresu
- **D-Bus (Desktop Bus)**
  - D-Bus (http://en.wikipedia.org/wiki/D-Bus) poskytuje aplikacím jednoduchý způsob vzájemné komunikace.
  - určitá forma zasílání zpráv dalším aplikacím přes centrální D-Bus uzel (proces, daemon)
  - využití hlavně pro desktopové aplikace v operačním systému Linux a dalších POSIXových OS.
  - aplikace si mohou zaregistrovat, jaké služby jsou nabízeny ostatním applikacím
  - součást projektu http://www.freedesktop.org/wiki/

Citováno z „http://wiki.zvesela.cz/index.php/Z%C3%A1kladn%C3%AD_a_strukturovan%C3%A9_formy_interakce_proces%C5%AF_a_vl%C3%A1ken.“

- Stránka byla naposledy editována 3. 6. 2011 v 11:46.

Sdílená pamět

Z Na státnice zvesela!

Oblast paměti sdílená více procesy

- vytvoření id = shmget(klíč, velikost, příznak)
- připojení skutečná_adresa=shmat(id,adresa,příznak) – adresa = kde chci začít, 0 → ať rozhodne kernel ; skutečná_adresa je vlastně pointer
- odpojení shmdt(skutečná_adresa)

Softwarově sdílená paměť

Pokud mluvíme o softwaru, rozumíme pod pojmem sdílená paměť metodu komunikace mezi procesy (IPC - Inter-process communication). Příkladem může být výměna dat mezi programy běžícími současně. Jeden z procesů si vytvoří prostor v RAM paměti, do kterého může druhý proces vstupovat.

Jelikož mohou oba procesy vstupovat do oblasti sdílené paměti jako do běžné paměti, jedná se o velice rychlý způsob komunikace (opak k ostatním mechanismům komunikace mezi procesy, jako jsou např. named pipes, Unix socket nebo CORBA). Nutno ovšem dodat, že tento způsob je méně výkonný, což je dáno právě tím že komunikace probíhá právě na jednom počítači, kdežto u ostatních IPC metodách může být ke komunikaci využita počítačová síť.

IPC prostřednictvím sdílené paměti se využívá především v Unixových systémech. POSIX poskytuje standardizované rozhraní pro programování aplikací (API - Application Programming Interface) pro využití sdílené paměti (POSIX Shared Memory).

Citováno z „http://wiki.zvesela.cz/index.php/Sd%C3%ADlen%C3%A1_pam%C4%9Bt“

- Stránka byla naposledy editována 16. 6. 2010 v 14:55.

Synchronizace v jádře, symetrický multiprocesing.

Z Na státnice zvesela!

Synchronizace v jádře

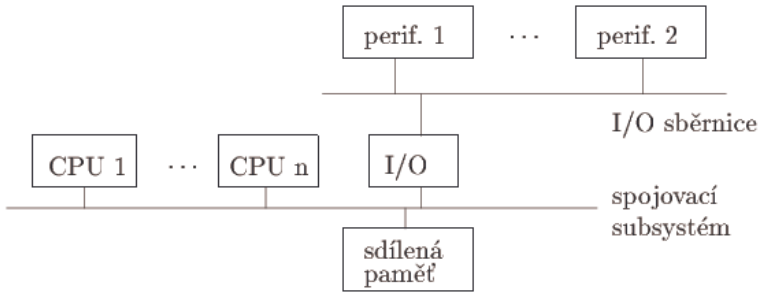
- **kritická oblast/sekce** - úsek kódu, který může být vykonáván současně nejvíce jedním procesem
- jádro sdílí datové struktury, vzniká tak problém synchronizace výpočtů v jádře
- **Metody synchronizace pro jednoprocesorové systémy**
  - **nepreemptivnost procesů v módu jádro** - pokud je proces běžící v módu jádro, nemůže se vykonat preempce, proces v módu jádro může být přerušen obsluhou výjimky nebo přerušení, neblokující systémová volání jsou atomická vzhledem k ostatním systémovým voláním
  - **atomické operace** - operace nad daty vykonané jedinou instrukcí jsou na HW úrovni atomické,
  - **zákaz přerušení** - maskování přerušení - data jádra, nad kterými pracují obsluhy přerušení můžeme efektivně zabezpečit tím, že při práci s nimi zakážeme přerušení -> kritické oblasti, vytvořené zákazem přerušení, musí být krátké, protože, když do nich jádro vstoupí, je blokována jakákoliv komunikace mezi V/V zařízeními a procesorem, metoda zákazu přerušení není použitelná, když se proces stane spícím
- **Metody synchronizace použitelné i pro víceprocesorové systémy**
  - **zamykání**
    - **jádrové semafory** – má položky – count (je-li kladná prostředek je volný), wait (uchovává adresu seznamu čekajících), waking (jenom jeden proces po uvolnění prostředku mohl tento získat), operace P() a V()
    - **kruhové blokování (spinlock)**
      - vlákno jednoduše čeká a ve smyčce pravidelně kontroluje, jestli byl zámek uvolněn.
      - Instrukce test-and-set zjistí hodnotu sdílené proměnné, 0(volno) nebo 1, a nastaví ji na hodnotu 1
      - neefektivní na jednoprocesorových systémech
      - spin_lock a spin_unlock můžou chránit jenom data jádra, ke kterým nikdy nepřistupují obslužné programy přerušení

```
spinlock_t TS(spinlock_t *lock) {
 /* začátek kritické sekce */
 spinlock_t loc = *lock;
 *lock = 1;
 return(loc);
 /* konec kritické sekce */
}
```

```
void spin_lock (spinlock_t *lock) {
 while (TS(lock) != 0) /*zamčeno*/
 ; /*cykluj*/
}

void spin_unlock (spinlock_t *lock) {
 *lock = 0;
}
```

SMP (symmetrical multiprocessing) architektura



- Procesory a sdílená hlavní pamět jsou připojeny ke společné sběrnici, tu je nutno zamykat pro výhradní přístup jednoho procesoru (aby nemohly 2 procesy zapisovat na stejné místo v paměti naráz).
- Tuto technologii používá většina dnešních víceprocesorových strojů.
- Nutná synchronizace mezipaměti (cache) procesorů: když procesor modifikuje svou mezipamět, musí kontrolovat jestli stejná data nejsou v mezipaměti jiného procesoru a když ano, musí je aktualizovat nebo zneplatnit.
- Díky SMP může být tedy úloha zpracovávána postupně různými procesory, protože, jak bylo řečeno, je umístěna ve sdílené hlavní paměti. To také přispívá k rovnoměrnému rozložení zátěže systému. SMP musí být ovšem podporováno operačním systémem, jinak ztrácí smysl.
- Kromě SMP existují další technologie pracující na podobném principu:
  - asymetrické multiprocesory ASMP - navíc vlastní lokální paměti a I/O připojení u procesorů, které tak mohou mít různé instrukční sady
  - multiprocesor s distribuovanou pamětí - procesory mají jen lokální paměti, není sdílená paměť, komunikace zasíláním zpráv, topologie 2D mřížky nebo N-rozměrné krychle, výhoda odstranění společné sběrnice jako úzkého hrdla
  - počítačové clustery - seskupení počítačů propojených počítačovou sítí, které spolu úzce spolupracují, takže navenek mohou pracovat jako jeden počítač

Řešení synchronizace na SMP:

- jádrové semafory
- kruhové blokování

Citováno z „http://wiki.zvesela.cz/index.php/Synchronizace_v_j%C3%A1d%C5%99e%2C_symetrick%C3%BD_multiprocesing.“

- Stránka byla naposledy editována 6. 8. 2010 v 13:41.

Atomické operace, synchronizační objekty a funkce

Z Na státnice zvesela!

Atomické operace

= operace, jejíž vykonávání je nepřerušitelné a má po celou dobu výhradní přístup ke svým datům

- operace nad daty vykonaná jedinou instrukcí jsou na HW úrovni atomické
- instrukce, které přistupují k paměti nejvíce jednou jsou atomické
- čti/modifikuj/zapiš instrukce, které čtou data z paměti, modifikují je a aktualizovaná data zapíšou zpátky do paměti jsou atomické, jestliže sběrnici mezi čtením a zápisem nezískal jiný procesor
- čti/modifikuj/zapiš instrukce, kterých instrukční kód má prefix lock jsou atomické i na multiprocesorových systémech, řídící jednotka zamkne sběrnici dokud instrucke není dokončena
- **TSL**
  - instrukce, která testuje hodnotu a nastaví paměťové místo v jedné nedělitelné (atomické) operaci
  - používá se pro testování a nastavení proměnné „zámek“ ve spin-locku
  - má tvar: *TSL R, lock* ...kde *R* je registr CPU, *lock* je proměnná (0,1 jako boolean)
  - má ji většina současných počítačů

Synchronizační objekty a funkce

Semafor

- Základní prostředek pro synchronizaci vláken
- Obsahuje frontu pro čekající vlákna, čítač a dvě funkce P(), V():
  - čítač – obsahuje nezáporné celé číslo, udávající, kolik procesů ještě může do KS vstoupit
  - P() – volá se před vstupem do kritické sekce (vláknem) - může vlákno blokovat dokud se semafor neuvolní
  - V() – volá se po výstupu z kritické sekce

Bariéra

- Prostředek pro synchronizaci více vláken v jeden okamžik
- Typicky se používá, pokud více vláken provádí výpočet a pro další pokračování výpočtu je nutné dokončení dílčí části u všech vláken. Vlákna, která skončí dílčí výpočet čekají na bariéře, dokud výpočet neukončení poslední vlákno, pak mohou opět všechna pokračovat v činnosti
- Zpravidla obsahuje čítač určující, kolik vláken se má ještě na bariéře zastavit. Dokud nedorazí všechna vlákna, všechny příchozí se uspí. Jakmile dorazí poslední vlákno, všechny vlákna čekající na bariéře se probudí

Binární semafor (mutex)

- Stačí pro vzájemné vyloučení přístupu k datům.
- Zaručí, že k datům může najednou přistupovat pouze jedno vlákno

Zámek

- Funguje podobně jako binární semafor, ale místo fronty uspaných vláken používá aktivní čekací smyčku
- např. Spin-lock s instrukcí TSL (Test ans Set Lock)

Monitor

- Objekt poskytující vláknům služby pro konkurenční prostředí
- Typový monitor – objektový typ (class), podle kterého se vytváří instance
- Model výpočtu pomocí monitoru
  - Vlákna se mezi sebou vůbec neznají, každé zná monitor
  - Veškeré interakce vláken probíhají nepřímo prostřednictvím monitoru
- Implementace monitoru obsahuje frontu, zámek, sdílená data procesů a služby monitoru (metody)
- Vlastnosti:
  - Volání metody (služby) musí být atomické (nejde přerušit z venku, musí být zajištěna konzistence dat)
  - Implementace atomicity vyžaduje zámek
  - Někdy nelze službu poskytnout -> uspat vlákno (blokující volání služby), nebo vrátit error code
  - Pro blokující volání mívají monitory podporu v podobě funkcí wait() (blokující) a notify(), notifyAll() (neblokující)
  - Je potřeba fronta čekajících vláken
  - Ve wait() je nutno odemknout zámek, za wait() zase zamknout (v monitoru smí být aktivní pouze jedno vlákno)
  - V Javě je jen 1 fronta vláken, ale ta mohou čekat z různých důvodů – volat notifyAll() a vlákna opět otestují možnost běhu
  - V POSIX vláknech je fronta implementována podmínkovou proměnnou

Rendezvous

- Používaná forma je synchronní (vlákna se při rendezvous synchronizují) a asymetrická (1 vlákno v roli klient, druhé v roli server – akceptuje rendezvous, klient musí znát (mít odkaz na) server, naopak ne).
- V každém vlákně se nachází jeho výkonný kód, v serveru se navíc nachází společný kód vláken, který se provede po zavolání entry (volá klient) a jeho akceptaci (provádí server)

Citováno z „http://wiki.zvesela.cz/index.php/Atomick%C3%A9_operace%2C_synchroniza%C4%8Dn%C3%AD_objekty_a_funkce“

- Stránka byla naposledy editována 6. 8. 2010 v 15:24.

Systém reálného času

Z Na státnice zvesela!

Dokončení výpočtu ve stanoveném termínu je kritické, termín musí být dodržen bez ohledu na zátěž.

RT systémy nejsou vysoko-výkonnostní → jde o to dopočítat včas, ne vypočítat co nejvíc.

Rozdělení

- **hard RT**
  - dokončení po termínu se považuje za chybu, výsledek je bezcenný
  - airbag, jaderná zařízení, řízení motoru,...
- **soft RT**
  - překročení termínu se toleruje – systém reaguje zhoršenou kvalitou služeb
  - např. zpracování obrazu

Plánování

- neosvědčil se cyklický plánovač

RMS (Rate Monotonic Scheduling)

- úlohám je přidělována statická priorita podle jejich periody = deadline
- úlohy jsou pak periodicky spouštěny za sebou podle přidělené priority
- snaží se dodržet deadlines všech úloh a splnit je v rámci jejich periody
- pro plánování využívá analýzu RMA, jinak více wiki (http://cs.wikipedia.org/wiki/Rate_monotonic_scheduling)

RMA (Rate Monotonic Analysis)

- přiřazuje statické priority úlohám tak, aby stihly výpočet v termínu
  - jedním z výstupů je zjištění, že to úloha nemůže stihnout
- periodická úloha
  - má opakovaně (periodicky) runnable v daných intervalech
  - priorita – kratší úlohy mají vyšší
  - plánování
    - Lui & Layland – pokud je zatížení procesoru n úlohami pod ln(2) (~70%), pak lze všechny naplánovat tímto static-priority algoritmem
    - pokud není 70%, pořád je možnost, že to lze → Response Time Test
- aperiodická úloha / sporadická úloha
  - např. zpracování událostí (HW přerušení), požadavek operátora
  - dělení:
    - soft deadline – změna nastavení radaru
    - hard deadline – pokyn pilota = řízení
  - plánování
    - v systému je několik procesů serverů, které vykonávají aperiodické/sporadické úlohy
    - servery se plánují jako periodické úlohy

- sporadické úlohy – setřídit podle deadline – EDF (Earliest Deadline First)

Příklady RTOS

- *PikeOS* - založený na mikrojádře a používá se převážně v embedded systémech a serverech, široké využití
- *RTLinux* - malý a rychlý operační systém, který je v souladu s normou POSIX pro minimální operační systémy reálného času
- *doplněk RTX pro Windows (Real-Time eXtension)* - zkracuje rozlišitelnou jednotku času z 5ms na 20mikrosekund, nezávislý plánovač vláken

Citováno z „http://wiki.zvesela.cz/index.php/Syst%C3%A9m_re%C3%A1ln%C3%A9ho_%C4%8Dasu“

- Stránka byla naposledy editována 9. 8. 2010 v 07:50.

Virtuální souborový systém

Z Na státnice zvesela!

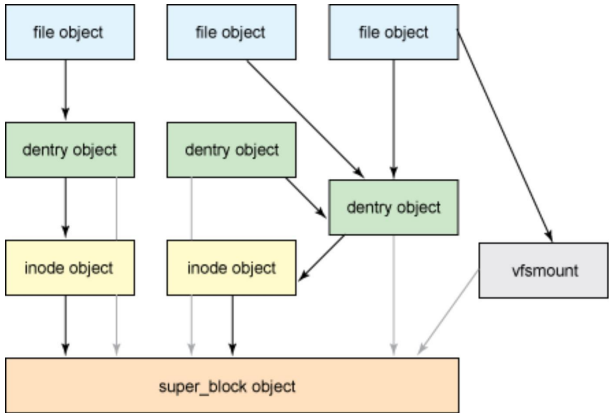
= vrstva jádra obsahující všechna systémová volání pro souborový systém, jednotné rozhraní pro různé souborové systémy → konkrétní implementaci poskytují drivery

- diskové FS
  - ext2, FAT, NTFS...
- síťové FS
  - NFS (Sun), SMB (Microsoft)
- speciální
  - devfs, procfs (linux) - nespravují diskový prostor

VFS objekty

- superblok – globální info o FS
  - typ FS
  - kořenový adresář
  - velikost bloku v bytech
  - operace
    - read_inode
    - write_inode
    - delete_inode
- inode – info o jednotlivém souboru, ID
  - číslo iuzlu
  - odkaz na dentry
  - počítadlo použití
  - odkaz na seznam iuzlů
  - id vlastníka
  - id skupiny
  - operace
    - create
    - lookup
    - mkdir
    - rmdir
- dentry – položka adresáře (např. pro /home/aaa jsou vytvořeny 3 dentry položky => / home aaa)
  - odkaz na sdružený inode
  - rodičovský adresář
  - odkaz na superblok
  - jméno souboru
  - operace
    - hash
    - compare
    - delete
- soubor – info o interakci mezi otevřeným souborem a procesem
  - ukazatel na dentry
  - mód (r,w,a)
  - pozice v souboru
  - operace

- read
- write
- llseek



Zamykání souborů

- přes systémové volání je možno zamknout část souboru (třeba i jeden byte) (vyžaduje POSIX)
- proces může vlastnit i několik zámků k jednomu souboru
- *poradní (advisory) zámky*
  - procesy musí spolupracovat - pokud je nějaká část souboru zamknuta a jiný proces nekontroluje její zamčení může jiný proces k zamčené části přistoupit

- *povinné (mandatory) zámky* (System V R3)
  - OS zařídí, že zámky se dodrží

- *sdílené zámky pro čtení* (může vlastnit více procesů)
- *výhradní zámky pro zápis* (může vlastnit jen jeden proces)

Další zdroje

Jak VFS vidí Šafařík (z toho je udělaná přednáška) (http://www.civilnet.cn/book/kernel/Understanding.the.Linux.Kernel(3rd%20Edition)/Understanding.the.Linux.Kernel(3rd%20Edition)/understandlk-CHP-12-SECT-1.html)

Jak VFS vidí kluci z IBM (http://www.ibm.com/developerworks/linux/library/l-virtual-filesystem-switch/index.html)

Jak VFS vidí soudruzi z NDR (pdf) (http://www.inf.fu-berlin.de/lehre/SS01/OS/Lectures/Lecture16.pdf)

Citováno z „http://wiki.zvesela.cz/index.php/Virtu%C3%A1ln%C3%AD_souborov%C3%BD_syst%C3%A9m“

- Stránka byla naposledy editována 9. 8. 2010 v 07:39.

Extended Filesystem

Z Na státnice zvesela!

- první verze operačního sytému Linux vycházely ze souborového sytému operačního systému Minix, později byl vytvořen Extended Filesystem – Ext FS a v roce 1994 byl uveden Ext2

Ext2

efektivnost

- při vytváření souborového systému je možné specifikovat velikost bloku (1024 až 4096 bytů), jestliže očekáváme soubory s několika tisíci bytů volíme velikost 1024, čím snižujeme interní fragmentaci (průměrně není využito půl bloku), na druhé straně veliké bloky pro rozsáhlé soubory snižují počet diskových operací

- pro diskovou oblast můžeme zadat povolený počet iuzlů podle očekávaného počtu souborů, co maximalizuje využití diskového prostoru
- souborový systém je rozdělen na skupiny bloků, každá skupina obsahuje bloky na sousedních stopách
- souborový systém předem přiřadí (preallocates) bloky obyčejným souborům, tj. předtím než jsou skutečně požadovány, při zvětšení souboru jsou tak dispozici sousedící diskové bloky
- podporuje rychlé symbolické odkazy, má-li symbolický odkaz 60 znaků nebo méně, je uložen v iuzlu

robustnost a flexibilita

- aktualizace souborů je navržena s ohledem na minimalizaci škody v případě havárie diskového systému - například při vytváření nového odkazu na soubor: napřed se zvýší počet odkazů v iuzlu a nové jméno se uloží do příslušného adresáře až následně
- podporuje automatickou kontrolu konzistence souborového sytému: při zavádění systému, po předdefinovaném počtu připojení souborových systémů, nebo po uplynutí předdefinovaného času od poslední kontroly
- podpora neměnných souborů
- podpora SVR4 i BSD sémantiky pro GID nového souboru
  - v SVR4 má nový soubor GID procesu, který ho vytvořil
  - v BSD nový soubor získá GID podle adresáře, ve kterém je vytvořen

Více info (http://cs.wikipedia.org/wiki/Ext2)

Ext3

- oproti ext2 navíc (zpětně kompatibilní):
  - žurnálování (informace o dokončených operacích) - možnost volby nejspolehlivějšího způsobu, kdy metadata i obsah souborů se ukládají do žurnálu a teprve poté jsou zapsány na disk (zápis 2x)
  - indexy souborů v adresáři implementované stromy (do té doby se používal pouze lineární seznam, v ext3 se používá jen na malé adresáře)

- možnost změnit velikost souborového systému za běhu (od listopadu 2004)

Ext4

- oproti ext3 navíc (zpětně kompatibilní):
  - umožňuje nasazení online defragmentátoru
  - Max. velikost oddílu 16 TB -> 1 EB
  - Max. velikost souboru 2 TB -> 16 TB
  - Max. počet podadresářů 32 768 -> neomezeno

Citováno z „http://wiki.zvesela.cz/index.php/Extended_Filesystem“

- Stránka byla naposledy editována 8. 8. 2010 v 14:32.

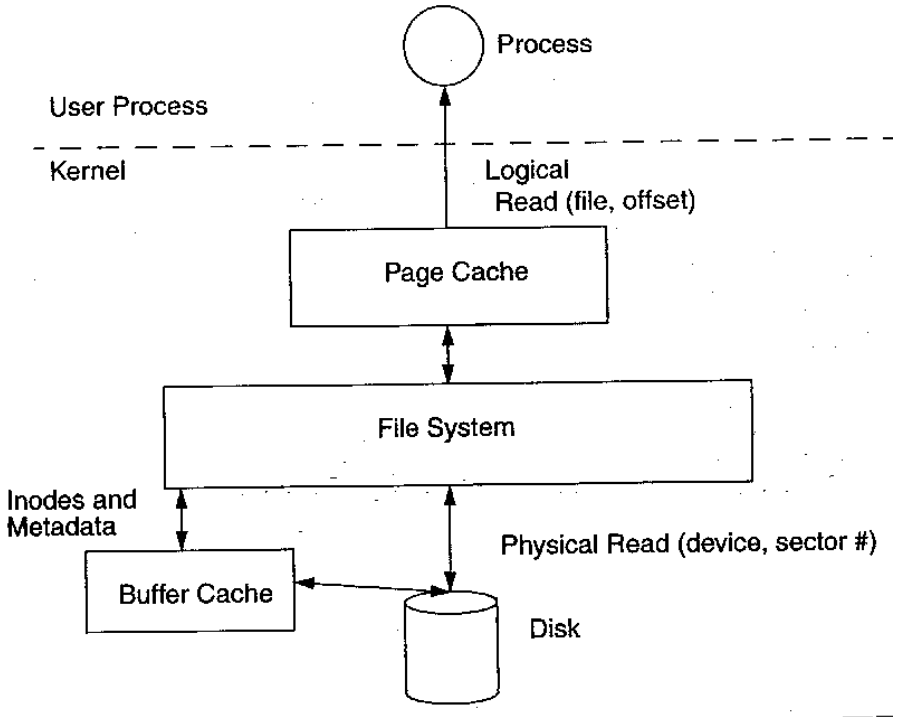
Správa V/V zařízení.

Z Na státnice zvesela!

- sdružení souboru a V/V zařízení - můžeme psát do souboru nebo poslat znak na tiskárnu
- každé zařízení má hlavní číslo (zařízení se stejným hl. číslem mají stejnou množinu operací -- stejný driver) a vedlejší číslo (identifikuje specifické zařízení)
- **soubor typu zařízení**: Trošku přehlednější ;-) (http://cs.wikipedia.org/wiki/Zařízení_(soubor))
  - **bloková zařízení** - přenášejí blok dat pevné velikosti jednou V/V operací, k blokům zařízení je libovolný přístup, mají buffer
  - **znaková zařízení** - přenášejí data různé velikosti jednou V/V operací, k datům je přístup sekvenční
  - **síťová zařízení** - nemají odpovídající soubor typu zařízení

- **práce VFS se soubory typu zařízení** - namísto souborových systémových volání požadované funkce jsou vykonány samotným zařízením, tj. jsou vykonávány driverem zařízení. V položkách tabulek chrdevs a blkdevs se nachází ukazatel na operace nad souborem typu zařízení, hlavní číslo zařízení je indexem do tabulky.

- **obsluha znakových zařízení** – data přímo do uživatelského adresového prostoru bez mezipaměti, podporováno mechanismem proudu (stream), což je plně duplexní zpracování a přenos dat mezi jádrovým adresovým prostorem driveru a uživatelským prostorem procesu
- **obsluha blokových zařízení**:
  - **V/V operace s vyrovnávací pamětí** - ve vyrovnávací paměti je uložen diskový blok, vyrovnávací paměti bloků jsou organizovány v mezipaměti vyrovnávacích pamětí bloků (block buffer cache) pomocí hlaviček bloků
  - **stránkové V/V operace** - přenos dat po blocích, adresový prostor procesu je množina stránek, V/V operace pro obyčejné soubory jsou vykonávány a ukládány po stránkách v mezipaměti stránek



Citováno z „http://wiki.zvesela.cz/index.php/Spr%C3%A1va_V/V_za%C5%99%C3%ADzen%C3%AD.“

- Stránka byla naposledy editována 12. 6. 2009 v 10:37.

Složitost, urychlení, účinnost a korektnost paralelního výpočtu.

Z Na státnice zvesela!

Obsah

- 1 Složitost (complexity)
- 2 Urychlení (speedup)
  - 2.1 Anomální urychlení
- 3 Účinnost paralelizace (efficiency)
- 4 Amdahlův zákon
- 5 Korektnost paralelizace

Složitost (complexity)

- je to funkce *f(n)*, jejíž hodnota je pro konkrétní algoritmus úměrná maximální době jeho výpočtu
- maximum se bere přes všechny možné vstupy algoritmu s rozměrem *n* (např. rozměr matice, počet čísel v paralelně realizovaném součtu apod.)
- např. sekvenční algoritmus pro součet *n* čísel má složitost *f(n) = n*, protože maximální doba výpočtu je úměrná počtu čísel *n*
- složitost se označuje písmenem *O*, v předchozím případě tedy *O(n)* znamená, že doba výpočtu je lineárně závislá na počtu čísel *n*.
- pro paralelní součet prováděný na *p = n / 2* procesorech je složitost *O(log n)*, kde logaritmus je s libovolným základem

Urychlení (speedup)

- označení *S*, obvykle se vyhodnocuje jako poměr doby výpočtu nejlepšího známého sekvenčního algoritmu a doby výpočtu paralelního algoritmu na témže (paralelním) počítači, využíváme-li *p* procesorů

Anomální urychlení

- při zběžném pohledu se zdá samozřejmé, že urychlení výpočtů nemůže být lepší než lineární podle počtu procesorů, tato úvaha však opomíjí, že více procesorů pohromadě může poskytovat nejen akumulaci výkonu, ale i jiných zdrojů
- vedle běžných, vcelku logických odlehčení, může např. rozdělení paměťového prostoru na více menších částí velmi omezit nutnost odkládání na disky, a tak lze někdy pozorovat i urychlení superlineární, tedy lepší než lineární.
- nejčastěji se tak děje ve dvou případech:
  - **Efekt cache-paměti** – Rozdělením výpočtu mezi více procesorů může dojít za příznivých podmínek k daleko častějšímu uplatnění lokálních cache-pamětí. Každý lokální výpočet je pak prováděn rychleji než v případě výpočtu jedním procesorem. Pokud algoritmus sám o

sobě vykazoval dobré urychlení, může pak být celkové urychlení lepší než lineární
- **Anomálie při prohledávání** – Paralelizované prohledávací algoritmy metodou „ořezávání“ pracují často rychleji, než by odpovídalo lineárnímu urychlení. Je to hlavně díky tomu, že paralelizovaný algoritmus je vlastně odlišný od sekvenčního a umožňuje většinou rychlejší upřesňování průběžného výběrového kritéria.

Účinnost paralelizace (efficiency)

- jedná se o urychlení dělené počtem použitých procesorů E = S / p
- např. nejlepší sekvenční algoritmus se počítá na uvažovaném počítači 10 s, výpočet hodnoceného paralelního algoritmu pro p = 4 procesory trvá 5 s. Urychlení je v tomto případě 2.0 a účinnost je 0.5.

Amdahlův zákon

- kvalitní paralelní algoritmy musí nepochybně vykazovat dostatečné urychlení při dostatečné účinnosti (tj. využití procesorů).
- dosažené urychlení je omezeno Amdahlovým zákonem, který bere v úvahu, že výpočet zpravidla nelze paralelizovat úplně, určitá část musí být provedena sekvenčně. Označíme-li tuto část *f*, pak pro výpočet probíhající na *p* procesorech je dosažitelné urychlení *S* limitováno podle vzorce:

$$S \le \frac{1}{f + \frac{1-f}{p}} \le \frac{1}{f}$$

- na první pohled to tedy vypadá, že i při sebevětším počtu procesorů nikdy nedosáhneme většího urychlení než *1/f*. Amdahlův zákon je však poplatný své době a obavy vychází z ne zcela oprávněných předpokladů:
  - byl uvažován pouze tradiční způsob postupné paralelizace. Při tomto způsobu jsou nejdříve objevena místa s největšími výpočetními nároky. Ty se pak víceméně tradičními postupy paralelizují. Ostatní místa se nechají provádět sekvenčně a jsou prohlášena za nenapravitelně sekvenční. Jistě, jejich paralelizace by možná vyžadovala větší úsilí, ale nelze ho vyloučit jednou provždy. Navíc pokud nějaká část úlohy musí probíhat sekvenčně, ještě to neznamená, že ostatní procesory musí zahálet.
  - druhým mnohdy zastřeným předpokladem je neměnnost podílu *1/f*. Použití paralelních počítačů umožňuje s rostoucím počtem procesorů řešení stále rozměrnějších úloh. Podle praxe se přitom zároveň ukazuje, že objem sekvenčních výpočtů narůstá velmi pomalu a podíl *1/f* se tak rychle zmenšuje.

Korektnost paralelizace

- požadavky na korektnost výpočtu jsou následující. Správný program je:
  - **Flow-correct** – program musí v každém běhu skončit a to vždy se stejným výsledkem
  - **Logically-correct** – Program navíc dává správný výsledek

Citováno z „http://wiki.zvesela.cz/index.php/Slo%C5%BEitost%2C_urychlen%C3%AD%2C_%C3%BA%C4%8Dinnost_a_korektnost_paraleln%C3%ADho_v%C3%BDpo%C4%8Dtu.“

- Stránka byla naposledy editována 8. 8. 2010 v 14:47.

Základní programové modely pro paralelizaci výpočetní činnosti (MPMD, SPMD, MPSD).

Z Na státnice zvesela!

Vlastnímu programování paralelní úlohy by měla předcházet fáze analýzy, ve které se rozhoduje o základním výpočetním přístupu, který se použije pro dekompozici výpočtu na jednotlivé složky - procesy.

MPMD (Multiple Program Multiple Data)

- několik procesorů autonomně vykonává více než jeden program nad různými daty
- jedná se o dekompozici výpočtu na relativně samostatné činnosti, z nichž některé mohou být vykonávány paralelně.
- tento přístup se využívá pro relativně složitou činnost a málo objemná data (tj. důraz je kladen na dekompozici činnosti, data potřebná pro dílčí činnost k ní pak logicky přiřadíme – označují se jako lokální data procesu, je snaha minimalizovat potřebu globálních dat, která nejsou přiřazena k žádnému konkrétnímu procesu)
- provedenou dekompozici lze obvykle vyjádřit precedenčním grafem, ve kterém hrany symbolizují činnosti a uzly potřebu synchronizace
- využitím modelu MPMD nemusí být sledováno pouze urychlení výpočtu. V mnoha případech (např. řízení v reálném čase, simulační programy) se tento model využívá s ohledem na lepší strukturování programu, kdy se každý proces stará „o to svoje“ a zároveň spolupracuje s ostatními.
- programový model MPMD lze implementovat jak na jednoprocesorovém počítači, tak na multiprocesorech různého typu.
- výsledkem je několik různých programů pracujících paralelně na různých datech
- lze vytvořit paralelní program např. v jazyce Ada na jednoprocesorovém počítači, odladit ho a pak přenést beze změny kódu (vše zařídí překladač) na symetrický multiprocesor. Na něm budou jednotlivé procesy probíhat fyzicky paralelně a výpočet bude rychlejší.

Např.:

- farmer-worker, kdy jeden proces úkoluje ostatní

SPMD (Single Program Multiple Data)

- několik procesorů autonomně vykonává jeden program nad různými daty
- tento model, označovaný jako dekompozice dat, se používá v případě, kdy relativně jednoduchá činnost je prováděna nad objemnými daty
- zpracovávaná množina dat (obvykle jednorozměrné nebo vícerozměrné homogenní pole prvků s jednoduchým typem) se v tomto případě rozdělí na *m* částí. Vytvoří se k procesů *(k <= m)* pracujících podle stejného programu a každý zpracuje jednu nebo několik strukturně podobných (ale hodnotami různých) částí dat
- je sledováno výhradně výkonnostní hledisko (urychlení výpočtu)
- už při analýze se uvažuje fyzicky paralelní výpočet a neuvažuje se, že by program mohl být

použit i na jednoprocesorovém stroji (kde by běžel pseudoparalelně a nedošlo by tedy k žádnému urychlení)
- průběh paralelizace (většinou cyklu) je následující: Různé iterace se svěří různým vláknům, nemá cenu zakládat větší počet procesů než je k dispozici procesorů (docházelo by k přepínání kontextu a tedy ke zpomalení), každé vlákno realizuje přibližně (počet iterací / počet vláken) iterací
- tento model je primárně využíván např. knihovnou MPI (kde je v zásadě realizovatelný i MPMD model výpočtu pomocí přepínače podle ID)

Použití:

- násobení matic - po prvcích, řádcích, či sloupcích
- iterační numerické řešení parciálních diferenciálních rovnic (geometrická dekompozice)

MPSD (Multiple Program Single Data)

- několik různých procesů zpracovává data v jednom datovém proudu
- jedná se o zřetězené zpracování dat, analogií z běžného života je montážní linka v továrně
- jedná se o zpracování rozsáhlého proudu datových prvků, přičemž nad jednotlivými prvky jsou vykonávány nějaké (libovolně složité) operace, které je možné svěřit různým specializovaným procesům a vykonávat je paralelně pro několik prvků datového proudu.

Použití:

- v tzv. Pipeline architektuře - např. grafická karta je zařízení optimalizované pro proudové zpracování dat
- pro výpočty odolné proti poruchám - několik různých systémů zpracovává ty samá data a musejí se shodnout na výsledku – např. řízení letu raketoplánu

- Stránka byla naposledy editována 8. 8. 2010 v 13:58.

Paralelizace cyklů

Z Na státnice zvesela!

Jedná se o případ *SPMD (Single Program Multiple Data)* - případ dekompozice dat.

Při analýze možné paralelizace cyklu je třeba určit jaké typy proměnných v cyklu jsou:

- lokální - inicializovány uvnitř smyčky
- sdílené - hodnoty se přenáší mezi iteracemi
  - nezávislé – když se pouze čtou
  - závislé
    - redukční – nejprve čtena, následně zapsána – ve stejné iteraci
    - uzamykané – čtena i zapisována v několika iteracích nebo několikrát v jedné - pokud by se iterace neprováděly sekvenčně, výsledek by byl stále OK
    - uspořádané – správný výsledek, jen když jsou iterace provedeny ve správném pořadí - již nelze urychlit tak, že vlákna současně vykonají operace nad svou částí pole (a pak jedno z nich zpracuje mezivýsledky)

Paralelizaci součtu pole lze provést snadno, protože se v cyklu vyskytuje nejvýše redukční proměnná, kterou stačí jen ošetřit mutexem.

Paralelizace součtu prefixů (výsledek pole součtů) již nelze provést jednoduchým rozdělením iterací vláknům, protože se v cyklu nachází uspořádaná proměnná. Musí se na to již použít docela sofistikovaný algoritmus (popsaný v přednášce PPR_3).

konkrétní příklady viz PPR přednáška 3 (http://wikiftp.zvesela.cz/PPR/PPR3_shared_spmd.pdf)

- Stránka byla naposledy editována 8. 8. 2010 v 17:09.

Programové prostředky pro multithreading: Java, rozhraní POSIX pro vlákna v jazyce C, podpora vláken ve WinAPI.

Z Na státnice zvesela!

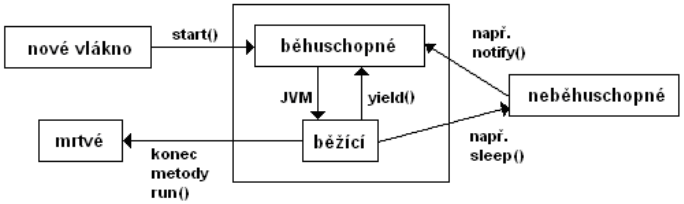
Obsah

- 1 Java
  - 1.1 Thread
  - 1.2 Monitory v Javě
- 2 POSIX
  - 2.1 Vlákna
  - 2.2 Mutexy (zámky)
  - 2.3 Podmínkové proměnné
- 3 WinAPI
  - 3.1 Process
  - 3.2 Thread
  - 3.3 Fiber
  - 3.4 Stavy vlákna
  - 3.5 Řízení běhu vlákna

Java

Thread

- tvoří základ všech paralelních programů v Javě
- pro jednoduché programy stačí oddědit od této třídy a překrýt metodu run(), do které se napíše výkonný kód vlákna.
- protože někdy je potřeba, aby naše třída dědila od jiné a zároveň měla vlastnosti vlákna, existuje ještě rozhraní **Runnable**, které stačí implementovat a třída rovněž získá vlastnosti vlákna
- pro napsání paralelního programu stačí vytvořit potřebné třídy, napsat jejich výkonné kódy do metod *run()* a pak vlákna spustit (vlákna se spouští metodou *start()*)
- stavy vlákna:
  - **Nové vlákno** – vlákno bylo vytvořeno, ale dosud nebylo spuštěno metodou start()
  - **Běhuschopné** – metoda start() už proběhla; těchto vláken může být více, ale na jednoprocesorovém stroji je vždy jen jedno běžící, ostatní musí čekat na předání řízení
  - **Neběhuschopné** – vlákno, které bylo uspáno metodou *sleep()*, nebo čeká na *wait()*, nebo čeká na I/O
  - **Mrtvé vlákno** – vlákno, jehož metoda run() již skončila



Monitory v Javě

- komunikace vláken je řešena přes sdílenou paměť
- kritické sekce jsou ošetřeny klíčovým slovem *synchronized*
- monitor je součástí každého objektu
- pro implementaci monitorů jsou uvnitř objektu monitoru skryté atributy:
  - **1 zámek** – pro všechny synchronizované metody
  - **1 fronta** – pro blokovaná vlákna čekající na vstup do synchronizované metody (krit. sekce)
  - nad frontou fungují privátní metody monitoru (objektu) *wait()*, *notify()* a *notifyAll()*
  - kromě těchto implicitních monitorů objektu lze v metodě ještě vytvořit synchronizační blok, který může použít i jiný zámek, než je implicitní zámek objektu, což dovoluje jemnější členění vzájemného vyloučení (např. jen na část metody)

POSIX

- jde o knihovnu pro jazyk C umožňující práci s vlákny.
- obsahuje tři základní typy objektů – vlákna (pthread_t), mutexy (pthread_mutex_t), podmínkové proměnné (pthread_cond_t), které jsou reprezentovány pomocí tzv. handle (zobecněný ukazatel)
- kromě handlů existují ještě tzv. atributované objekty (k popisu vlastností vláken, mutexů, podmínkových proměnných) pthread_attr_t, pthread_mutexattr_t, pthread_condattr_t.
- vytváření a rušení objektů je dynamické, každé vlákno má svůj zásobník, tj. proměnné definované v programu vlákna jsou lokální.
- proměnné definované v hlavním programu jsou globální (sdílené a musejí se zamykat)

Vlákna

- program vlákna je vlastně funkce C s typem *void* prog_name(void* arg)*
- vlákno se vytvoří následujícím způsobem:

```
int status;  //0 uspech, jinak chyba
status = pthread_create(&worker, NULL, prog_name,…);
```

- NULL znamená ukazatel na atributy objektu, pokud je to NULL, vytvoří se automaticky atributový objekt s implicitním nastavením
- vlákno běží hned po vytvoření
- stavy vlákna jsou ready, running, waiting a terminated
- vlákno se ruší pomocí detach
- **Atributy**:
  - způsob plánování: **FIFO** – nejprioritnější kategorie (nejdříve se plánují vlákna této kategorie), **RR** – round-robin, **FG** – foreground, implicitní kategorie, střídání vláken, ty s vyšší prioritou mají více času, **BG** – background, všechna vlákna se střídají, ale dostávají

méně času než FG
- priorita
- rozměr zásobníku
- hlídač zásobníku – jak daleko se můžeme od konce zásobníku dostat, jinak vznikne výjimka
- konec vlákna - **vlákno dojde na konec svého programu** - na toto ukončení se lze synchronizovat z jiného vlákna pomocí *pthread_join(na_koho_se_čeká, kam_přijde_výsledek)* nebo **zabito z vnějšku** – pokud možno nepoužívat, základní funkce pro likvidaci *pthread_cancel(oběť)*; oběť se může bránit *pthread_setcancelstate(...)*, k likvidaci nemůže dojít kdekoliv v kódu vlákna, ale jen v předem připravených místech (volání blokující funkce, volání *pthread_testcancel()*), popisovaný způsob je synchronní, existuje i asynchronní

Mutexy (zámky)

- exitují k ochraně globálních dat, ne pro synchronizaci- k synchronizaci lze vyrobit semafory
- většinou implementovány jako busy-wait (je-li zámek zamčen vlákno se na něm zacyklí, tj. spotřebovává prostředky a je běžící či připravené k běhu)
- hodí se pro krátké kritické sekce z výše uvedeného důvodu
- vytvoření mutexu:

```
pthread_mutex_t zamek = PTHREAD_MUTEX_INITIALIZE;
```

- tři typy zámků:
  - normální – konkrétní vlákno ho může zamknout jen 1x
  - rekurzivní – konkrétní vlákno ho může zamknout vícekrát (pro rekurzivní zpracování globálních dat)
  - ladící – *ERRORCHECK*; Pozná se opakované zamčení
- Základní funkce:
  - *pthread_mutex_lock(&zamek);*
  - *pthread_mutex_unlock(&zamek);*
  - *pthread_mutex_try_lock()* – neblokující zamykání

Podmínkové proměnné

- implementují frontu, kde vlákna *pasivně* čekají na splnění podmínky, neimplementují test podmínky (ten musím být v kódu vlákna).
- aby test podmínky a případná změna proměnné byla atomická akce, je třeba sdružit podmínkovou proměnnou s mutexem
- typy podmínkových proměnných:
  - *pthread_cond_wait(&podm_prom, &mutex);* - čekající vlákno musí odemknout mutex
  - *pthread_cond_signal(&podm_pom);* - jako *notify()*
  - *pthread_cond_broadcast(&podm_prom);* - jako *notifyAll()*

WinAPI

- preemptivní multitasking a preemptivní multithreading

Process

- bezpečnostní kontext = kdokoliv nemůže provádět cokoliv, jedinečný identifikátor, spustitelný

kód
- prioritní třída – vlákna mohou mít prioritu pouze v rozsahu prioritní třídy procesu
- má alespoň jedno, primární, vlákno, které může vytvářet vlákna – threads a fibers

Thread

- entita v rámci procesu, kterou plánuje OS
- priorita
- všechny vlákna sdílí paměťový prostor a zdroje svého procesu
- může mít vlastní bezpečnostní kontext – možnost převzetí role někoho jiného (impersonating)

Fiber

- běží v kontextu vlákna, plánuje ho thread procesu, jeden thread může naplánovat několik fibers
- má zásobník, má menší kontext, má přístup do TLS threadu v jehož kontextu běží, nemá prioritu

Stavy vlákna

- Initialized — sjelo z výrobní linky jádra OS
- Ready — připraveno ke spuštění na některém z procesorů
- Running — běží
- Standby — bude spuštěno, vždy pouze jen jedno vlákno
- Terminated — RIP
- Waiting — není připraveno běžet, až bude, bude naplánováno
- Transition — vlákno čeká na něco jiného než je procesor
- Unknown — (klientu WMI) Neznámý stav

Řízení běhu vlákna

- CreateThread
- CreateRemoteThread - v adresovém prostoru jiného procesu
- Suspend, Resume
- Kill, Terminate, Exit
- Sleep
- Get/SetPriority

Thread Pool - množina vláken, která zpracovává asynchronní události pro process, místo vytváření a rušení vláken, která běží jen krátkou dobu

- Stránka byla naposledy editována 8. 8. 2010 v 18:17.

Konstrukce jazyka Ada pro paralelní programování.

Z Na státnice zvesela!

- Ada je OO jazyk se silnou verifikací typů, nepoužívá interpretr, nepodporuje fibres
- paralelní části výpočtu se označují jako tasky
- mohou být prováděny na jednom procesoru, více procesorech nebo více počítačích, není to však vyjádřeno v kódu programu
- pro synchronizaci tasků se používá asymetrické synchronní rendezvous (zasílání zpráv)

Tasky

- konstrukce *task* představuje program paralelně proveditelného procesu, schopného komunikace s ostatními procesy
- deklarace je následující:

```
task jméno is
       deklarace jmen komunikačních typů
end jméno;
task body jméno is
       lokální deklarace a příkazy
end jméno;
```

- pro ukončení procesu lze použít příkaz *abort jméno;*, ale jeho použití by mělo být výjimečné
- pěkný popis tasků atd. je na [wikibooks (http://en.wikibooks.org/wiki/Ada_Programming /Tasking)]

Rendez-vous

Pro interakci procesů používá ADA principu asymetrického rendezvous, kterým eliminuje potřebu semaforů (umí je nahradit), umožňuje synchronní a nepřímo (zavedením pomocného procesu) i asynchronní komunikaci procesů zasíláním zpráv. Dovoluje tak elegantní konstrukci monitorů.

- prostředek pro synchronizaci úkolů (tasks)
- dva úkoly spolu komunikují pomocí rendez-vous - Meeting point, entry calls
- task je uspán do té doby, než se dostaví druhý task, který s ním chce komunikovat

```
task Simple_Task is
  entry Start(Num : in Integer);
  entry Report(Num : out Integer);
end Simple_Task;

task body Simple_Task is
  Local_Num : Integer;
  begin
  //čeká na vložení čísla - entry call
  accept Start(Num : in Integer) do
    Local_Num := Num;
  end Start;
  //normálně pokračuje v běhu
  Local_Num := Local_Num * 2;
  //čeká na vyzvednutí spočítané hodnoty
  accept Report(Num : out Integer) do
    Num := Local_Num;
  end Report;
end Simple_Task;
```

- uvedený příklad stačí, pouze pokud potřebujeme jen jedno vlákno běžící podle uvedeného kódu
- průběh dostaveníčka - accept:
  - klient zavolá server
  - server si převezme parametry
  - server provede výpočet, klient spí
  - server předá výsledky
- **Select** - může být nezbytné, aby úkol mohl reagovat na několik vstupních volání (entry calls) – pokaždé na jiné dle okolností – tj. ne v předem určeném pořadí

```
//Vynutíme si inicializaci a další se
//už pak může vykonávat v libovolném
//pořadí.
accept Init(Item : in Integer) do
  Local_Item := Item;
end Init;

loop
  select
    accept Stop;
      exit;
  or

  when podmínka = > //může i nemusí být
    accept Put(Item : in Integer) do
      Local_Item := Item;
    end Put;
    Local_Item := Local_Item * 2;

  else
    Put_Line("No entry call at this time");
  end select;

  delay 0.01;
end loop;
```

- **Protected Objects, Protected Types**
  - tasky mohou sdílet objekty
  - objekt je instance typu – klíčové slovo *type*
  - klíčové slovo *protected* zajistí exkluzivní přístup k chráněnému objektu
  - jsou tři operace nad chráněnými objekty:
    - Procedury – mění stav objektu, aniž by pro to musela být splněna podmínka; překladač se stará, aby měly exkluzivní přístup k objektu
    - Entry calls – stejné jako procedury, ale pro vykonání entry call je třeba navíc splnit podmínku
    - Funkce – pouze vrací stav a nic nemění a proto nemusí mít exkluzivní přístup k objektu

- Stránka byla naposledy editována 13. 8. 2010 v 15:52.

Výpočetní prostředí s distribuovanou pamětí – charakteristika a principy realizace základních modelů paralelního výpočtu.

Z Na státnice zvesela!

Výpočetní prostředí s distribuovanou pamětí můžeme rozdělit na dva typy:

- **Počítačové sítě (též vícepočítačové či multipočítačové systémy)**
  - jsou vytvořeny spojením několika (většinou různých) počítačů komunikačními linkami
  - jsou místně rozlehlé
  - při použití vhodného programového prostředku pro spojení počítačů se pak každá stanice může tvářit jako jeden procesor multiprocesorového stroje
  - pro tento přístup se nečastěji využívá PVM
- **Paralelní počítače**
  - jsou vytvořeny spojením většího počtu výpočetních a dalších prvků do jednoho funkčního celku
  - jsou místně kompaktní (např. v jedné skříni) a zpravidla homogenní (výpočetní prvky stejného typu)
  - pro psaní paralelních programů pro tento typ se často využívá MPI

- protože neexistuje sdílená paměť, používá se pro komunikaci mezi procesy především zasílání zpráv
- protože systémy s distribuovanou pamětí nemají žádný úzký profil ve formě sběrnice, přes kterou by procesory přistupovaly ke sdílené paměti, hodí se pro úlohy vyžadující tzv. masivní paralelismus (stovky až tisíce procesorů)

HW pohled na architekturu

- topologie obecně
  - pravidelná (mřížka, krychle)
  - nepravidelná (Internet)
- topologie fyzická
  - pevná (mřížka, krychle)
  - flexibilní (packet switching, circuit switching)
- směrování
  - pouze mezi sousedy - při pevné topologii
  - podle příjemce - známe z IP
  - podle odesilatele - odesílající si určí cestu
- fyzická adresa uzlu
  - měla by souviset s polohou uzlu v síti
  - mělo by z ní možné být odvodit adresy sousedů

SW pohled na architekturu

- alokace uzlů
  - 1 proces 1 uzel
  - více procesů 1 uzel
  - celá síť jeden výpočet
  - část sítě jeden výpočet
  - celá síť více výpočtů
- identifikace uzlů
  - 1 proces 1 uzel -> adresa procesu = adresa uzlu
  - více procesů 1 uzel -> musíme být schopni rozlišit procesy na uzlu
  - migrující proces -> nutné udržovat tabulku proces/uzel, nebo přenechat na middleware
- topologie
  - fyzická
  - síťová
  - virtuální
  - ideálně fyzická = síťová = virtuální = zároveň nejlepší možnost

Základní modely

SPMD

- nejčastěji používaný model v prostředí s distribuovanou pamětí
- do procesorů se zavede množina procesů pracujících podle jednoho programu a každý dostane ke zpracování část objemných dat (části dat jsou strukturně stejné, ale hodnotami různé)
- procesy si vyměňují informace (většinou zasílání dílčích výsledků) zasíláním zpráv
- většinou se používá přístup **farmer-workers**, tedy jeden řídící proces a *n* dělníků
  - řídící proces rozdělí úlohu na části, které předá dělníkům a následně pak agreguje dílčí a vytváří globální výsledky
  - dělníci dostanou přidělen úsek práce a vytváří dílčí výsledky, které předávají řídícímu procesu
  - program hlavního procesu a dělníka může být oddělen do dvou samostatných souborů (typicky PVM), nebo jeden zdrojový soubor obsahuje jak program dělníka, tak hlavního procesu a určení, co má který proces dělat, se provádí až během výpočtu (typicky MPI)
- problém rozdělování práce
  - rozdělování práce mezi procesy je možné buď staticky podle počtu použitých procesorů, nebo dynamicky.
  - při statickém rozdělení se vytvoří tolik procesů, kolik je k dispozici procesorů a každý proces dostane stejný kus práce. Problémem je, obzvláště při použití PVM, neznámé zatížení jednotlivých procesorů a výsledek je tedy znám až po skončení procesu na nejpomalejším procesoru
  - při dynamickém rozdělování se vytvoří více pracovních jednotek než je procesorů (tyto jednotky jsou ve srovnání se statickým způsobem mnohem menší) procesorům (počítačům) jsou dynamicky přidělovány podle potřeby, tj. když zpracují předchozí jednotku. Výhodou je nezávislost na nejpomalejším procesoru, nevýhodou je zřejmá komunikační režie výpočtu. Přirozeně dochází k *load-balancingu*, výkonnější stroje zpracují více jednotek.

MPMD

- v tomto případě existují různé procesy pracující podle různých programů nad strukturně jinými daty
- procesy mezi sebou komunikují zasíláním zpráv
- do tohoto modelu lze zařadit i řetězové zpracování **MPSD**, při kterém si procesy probíhající podle různých programů „předávají“ ke zpracování datové záznamy
  - problémem může být kapacita přenosových kanálů -> řeší se překrytím doby komunikace dobou výpočtu

Výpočetní prostředí s distribuovanou pamětí – charakteristika a prin... http://wiki.zvesela.cz/index.php/Výpočetní_prostředí_s_distribuova...

Citováno z „http://wiki.zvesela.cz/index.php/V%C3%BDpo%C4%8Detn%C3%AD_prost%C5%99ed%C3%AD_s_distribuovanou_pam%C4%9Bt%C3%AD_%E2%80%93_charakteristika_a_principy_realizace_z%C3%A1kladn%C3%ADch_model%C5%AF_paraleln%C3%ADho_v%C3%BDpo%C4%8Dtu.“

- Stránka byla naposledy editována 8. 8. 2010 v 19:25.

Charakteristika a porovnání výpočetních nástrojů PVM a MPI, příklady použití.

Z Na státnice zvesela!

- PVM i MPI se používají v prostředí s distribuovanou pamětí

PVM (Parallel Virtual Machine)

- PVM je v nejobecnějším pohledu univerzální výpočetní model pro paralelní programování, který má dobrou naději stát se mezinárodním standardem
- je implementováno pro paralelní počítače různého typu (tj. z pohledu programátora se jedná o programovací prostředek) a je k dispozici ve formě knihoven v programovacích jazycích C, Fortran a Java.
- uplatňuje se především v počítačových sítích, kdy z různorodých propojených počítačů se vytvoří virtuální multiprocesorový počítač
- jedná se o poměrně nízkoúrovňový nástroj používající pro vzájemnou interakci procesů asynchronní zasílání zpráv přes vyrovnávací paměti
- každý proces má jednu aktivní vyrovnávací paměť pro vysílání a jednu pro čtení - vytváří se a ruší dynamicky
- klíčovou částí prostředí PVM je proces PVMD běžící na pozadí (démon) na každém počítači, který je zařazen do virtuálního multiprocesoru vytvořeného pro konkrétního uživatele
- počítače použité v konfiguraci virtuálního multiprocesoru jsou v rámci PVM identifikovány svým síťovým jménem
- na jednom stroji může být alokováno několik procesů jedné aplikace i několik démonů PVMD (pro jinou aplikaci)
- k manuálnímu ovládání virtuálního multiprocesoru je k dispozici PVM konzola
- příkazy konzole:
  - add host_name – Přidá stroj se síťovým jménem host_name do konfigurace virtuálního multiprocesoru
  - spawn – spustí výpočet připravené aplikace
  - conf – Vypíše konfiguraci virtuálního multiprocesoru (jména typ strojů, identifikátory procesů PVMD,…)
  - mstat host_name – Vypíše stav specifikovaného stroje
  - pstat proces_id – vypíše stav specifikovaného procesu
  - ps –a - vypíše všechny procesy běžící aplikace, jejich alokaci na stroj a identifikátory
  - sig signal_num – Pošle signál procesům, lze je pomocí toho příkazu takto ovládat
  - kill process_id – Ukončení libovolného procesu aplikace
  - quit – Ukončí výpočet
- pro paralelizaci výpočtu existuje množství metod, všechny mají předponu pvm_
- paralelní program je typicky rozdělen na hlavní proces, který rozděluje práci a řídí celý výpočet a dělníky, kteří provádějí dílčí výpočty
- řízení dělníků a předávání dat se děje pomocí zasílání zpráv

MPI (Message Passing Interface)

- MPI je knihovna pro podporu paralelních výpočtů s systémech s distribuovanou pamětí
- je k dispozici pro jazyky C, Fortran i Java; všechny funkce knihovny mají předponu MPI_
- není zaručen determinismus chování programu a překladač může provádět jen velmi omezené kontroly správného využití MPI funkcí
- oproti PVM se jedná o programovací prostředek vyšší úrovně, vztah mezi nimi je asi jako mezi jazykem symbolických adres (PVM) a vyšším programovacím jazykem (MPI)
- lze říci, že v PVM lze naprogramovat "skoro cokoliv", ale dá to velkou práci a program bude patrně nepřenositelný
- dominantní aplikací pro MPI jsou numerické výpočty s regulárními daty (vektory či matice s číselnými prvky), přičemž pro takové aplikace je programování relativně pohodlné a program je dobře přenositelný na jiné instalace MPI
- další vlastnosti MPI můžeme shrnout do několika bodů:
  - MPI je primárně určeno pro homogenní výpočetní prostředí, takže poskytuje prostředky i pro "synchronní" algoritmy (tj. takové, kde se předpokládá přibližně stejná rychlost běhu jednotlivých procesů) a odpovídající statické rozdělení práce mezi procesy
  - MPI primárně využívá SPMD model paralelního výpočtu, tj. vyrobí se, na rozdíl od PVM, jen jeden spustitelný soubor programu a ten se zavede do zvoleného počtu procesorů
  - komunikace procesů je asynchronní zasílání zpráv s přímým adresováním přes číselné ID procesu
  - MPI má svoje **primitivní datové typy** a z nich lze skládat **strukturované typy**.
  - oproti PVM zde existuje sada funkcí pro tzv. **globální operace**, tj. operace nad daty, jejichž instance jsou rozprostřeny ve všech procesech výpočtu
- přestože se vytváří jen jeden spustitelný soubor, program je většinou, podobně jako v PVM, členěn na hlavní proces rozdělující práci a agregující dílčí výsledky a pracovní procesy, které počítají dílčí výsledky
- členění je provedeno přímo v programu, který tedy musí obsahovat jak kód hlavního procesu, tak dělníků - rozlišení se provádí podle čísla procesu (každý proces má unikátní číslo, procesy jsou číslovány od nuly a typicky 0 je hlavní proces)
- protože MPI je primárně určeno pro zpracování rozměrných polí a matic dat, obsahuje globální funkce pro rozdělení dat mezi pracovní procesy (Bcast,Scatter) i funkce pro snadné získání globálních výsledků (Reduce,Allreduce,Gather)
- na rozdíl od PVM se používá spíše na fyzických multiprocesorech, přičemž počet procesorů použitých ve výpočtu se zadává při spuštění programu

Ukázka programu který po spuštění zjistí počet procesů a vypíše jej, pouze jednou.

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

int main(int argc, char *argv[]){
  int pocet = 0;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(COMM_WORLD, &pocet);
  MPI_Comm_rank(COMM_WORLD, &moje_id);
  if (moje_id == 0)
    printf("Pocet procesu: %d",pocet);
  MPI_Finalize();
}
```

Citováno z „http://wiki.zvesela.cz/index.php/Charakteristika_a_porovn%C3%A1n%C3%AD_v%C3%BDpo%C4%8Detn%C3%ADch_n%C3%A1stroj%C5%AF_PVM_a_MPI%2C_p%C5%99%C3%ADklady_pou%C5%BEit%C3%AD.“

- Stránka byla naposledy editována 9. 8. 2010 v 19:18.

Překladače – typy, struktura a princip činnosti.

Z Na státnice zvesela!

Překladač – obvykle program, který čte zdrojový program a převádí ho do cílového jazyka, zdrojový program je napsaný ve zdrojovém jazyce, cílový program v cílovém jazyce, důležitou částí překladače jsou diagnostické zprávy (informování o chybách ve zdrojovém programu)

Typy překladačů:

- **Křížový překladač (Cross Compiler) -** je to překladač generující cílový program pro jiný druh počítače, než na kterém je prováděn překlad. Hlavní důvod použití takového překladače je malý paměťový prostor na cílovém počítači (kompilátor se tam nevejde).

- **Silikonový překladač (Silicon compiler (http://en.wikipedia.org/wiki/Silicon_compiler))**
  - silikonové překladače pracují pouze s logickými hodnotami a užívají se k návrhu logických obvodů.
  - převádí jazyky popisující hardware (např. Verilog, VHDL) do logických obvodů

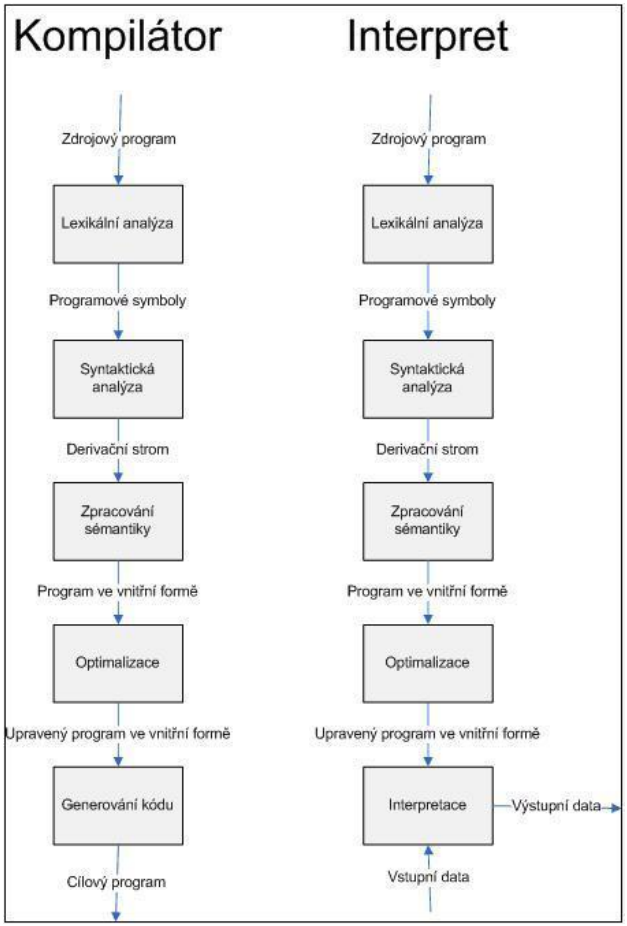
- **Formátory textu** - jde o úpravu textu podle požadavků uživatele, např. TeX. Např. syntax highlighting, nebo přeformátování kódu – odsazení apod. Takový překladač, který ze vstupního souboru definovaného určitým jazykem, vygeneruje výstup.

- **Kaskádní překladač** - kaskádní kompilátor použijeme v případě, máme-li vytvořit překladač z jazyka A do jazyka C a je-li k dispozici již kompilátor z jazyka B do jazyka C. Pak vytvoříme kompilátor z jazyka A do jazyka B, pokud je to snazší než vytvořit kompilátor z jazyka A do jazyka C. Jazyk B je vnitřním jazykem, a pokud je to standardní všeobecně používaný jazyk, pak programy v jazyce A budou snadno přenositelné. Nevýhodou je však, že oba překladače produkují chybové zprávy. Chybové zprávy druhého překladače jsou cizí pro uživatele jazyka A, protože jsou orientovány na jazyk B.

Dva hlavní typy překladačů:

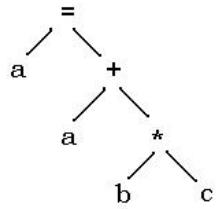
- **Kompilátor** – všechny příkazy překládá najednou, program lze spustit až po ukončení celého překladu (Pascal, Java, C, Fortran, Ada, …)

- **Interpret** – zpracovává příkazy jednotlivě a každý provede okamžitě po jeho přeložení (Python, Perl, JavaScript, Ruby, …)



Lexikální analýza – zdrojový kód vstupuje do procesu překladu jako posloupnost znaků. Tato posloupnost se čte lineárně zleva doprava a sestavují se z ní lexikální symboly jako konstanty, identifikátory, klíčová slova nebo operátory. Je založena na regulárních gramatikách. Výsledkem je posloupnost symbolů, např. je na vstupu rozeznáno klíčové slovo begin a do posloupnosti lexikálních symbolů bude zařazen nový symbol reprezentující právě toto klíčové slovo. Tyto symboly jsou programem snadno použitelné a dále zpracovatelné. V této fázi se odstraňují veškeré komentáře.

Syntaktická analýza – Z posloupnosti lexikálních symbolů se vytvářejí hierarchicky zanořené struktury, které mají jako celek svůj vlastní význam, např. výrazy, příkazy, deklarace nebo program. Programy jsou psány většinou v infixové notaci (a=a+b*c) => analyzujeme a vytváříme hierarchické uspořádání derivačního stromu:



Sémantická analýza – provádějí se některé kontroly, zajišťující správnost programu z hlediska vazeb, které nelze provádět v rámci syntaktické analýzy (např. kontrola deklarací, typová kontrola, apod.).

Optimalizace – Optimalizátor kódu zajišťuje, aby se používalo co nejméně pomocných proměnných pro mezivýpočty, aby se v cyklu zbytečně několikrát nevyhodnocoval tentýž výraz, jestliže hodnota jeho prvků zůstává bez změny a vyhodnocení stačí provést jednou před cyklem, apod. Optimalizací prochází program obvykle v intermediálním tvaru – intermediální kód je již podobný cílovému programu, má však strukturu vhodnější pro optimalizaci. Může to být zápis podobný assembleru nebo třeba dynamická struktura (dynamický seznam stromů představujících jednotlivé příkazy).

Generování kódu – poslední fází překladače je generování cílového kódu, což je obvykle přemístitelný kód nebo program jazyka asembleru. Všem proměnným použitým v programu se přidělí místo v paměti. Potom se instrukce mezikódu překládají do posloupnosti strojových instrukcí, které provádějí stejnou činnost.

Citováno z „http://wiki.zvesela.cz/index.php/P%C5%99eklada%C4%8De_%E2%80%93_typy%2C_struktura_a_princip_%C4%8Dinnosti.“

- Stránka byla naposledy editována 10. 8. 2010 v 14:53.

Regulární gramatiky, regulární výrazy a konečné automaty.

Z Na státnice zvesela!

Gramatika - Gramatika G je čtveřice (N,Σ,P,S), kde:

- N je konečná množina neterminálních symbolů (neterminálů).
- Σ je konečná množina terminálních symbolů tak, že žádný symbol nepatří do N a Σ zároveň.
- P je konečná množina odvozovacích pravidel. Každé pravidlo je tvaru

$$(\Sigma \cup N)^* \longrightarrow (\Sigma \cup N)^*$$

- *S je prvek z N nazývaný počáteční symbol.*

Regulární gramatika – je to gramatika typu 3 = lineární, navíc převedená do regulárního tvaru (podle Chomského hierarchie). Pravidla těchto lineárních gramatik jsou omezena na jeden neterminál na levé straně. Pravá strana se u pravé regulární gramatiky skládá z jednoho terminálu (u lineární i z více), který může být následován jedním neterminálem, tedy:

$X \to wY$
$X \to w,$

Obdobně se definují i levé regulární gramatiky, které obsahují pravidla typu:

$X \to Yw$
$X \to w$

Pravé a levé gramatiky jsou ekvivalentní. Jazyky generované regulárními (=lineárními) gramatikami jsou právě jazyky rozpoznatelné konečným automatem.

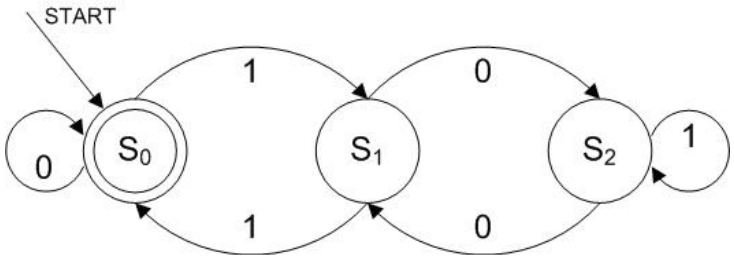
Regulární výraz - je řetězec popisující celou množinu řetězců, konkrétně regulární jazyk. Používají se nejčastěji v počítačových programech a skriptovacích jazycích pro vyhledávání a úpravu textu. V případě, že uživatel chce v textu vyhledat nějaký řetězec, který nezná přesně nebo který může mít více variant, může zadat regulární výraz, který postihne všechny chtěné varianty. Program tak nalezne všechny části textu, které danému výrazu odpovídají.

Konečný automat – formálně je konečný automat definován jako uspořádaná pětice (S,Σ,P,s,F) , kde:

- S je konečná množina stavů.
- Σ je konečná množina vstupních symbolů nazývaná abeceda.
- P je tzv. přechodová funkce (též přechodová tabulka), popisující pravidla přechodů mezi stavy.
- s je počáteční stav (s naleží S)
- F je množina koncových stavů (F náleží S)

Popis činnosti automatu: Na počátku se automat nachází v definovaném počátečním stavu. Dále v každém kroku přečte jeden symbol ze vstupu a přejde do stavu, který je dán hodnotou, která v

přechodové tabulce odpovídá aktuálnímu stavu a přečtenému symbolu. Poté pokračuje čtením dalšího symbolu ze vstupu, dalším přechodem podle přechodové tabulky, atd. Podle toho, zda automat skončí po přečtení vstupu ve stavu, který patří do množiny koncových stavů, platí, že automat buď daný vstup přijal nebo nepřijal. Množina všech řetězců, který daný automat přijme, tvoří regulární jazyk.



Meze regulárních gramatik

Jak rozpoznat zdali je nějaký jazyk možné sezobnout regulárním výrazem (konečným automatem, regulární gramatikou)? K tomuto lze použít tzv. Pumping teorém [wiki (http://en.wikipedia.org/wiki/Pumping_lemma_for_regular_languages)] nebo česky [abclinuxu (http://www.abclinuxu.cz/clanky/programovani/jazyky-a-prekladace-1-uvod#meze-regularnich-jazyku)].

Teorém vlastně říká, že v dostatečně dlouhém slově *w* daného regulárního jazyka můžeme nalézt tři části — *x*, *y* a *z*, přičemž nejdůležitější část *y* může zahrnovat i celé slovo. Aby byl tento jazyk regulární, musí platit, že část *y* můžeme ze slova vyjmout, nebo jí libovolně zopakovat, a přitom stále zůstáváme v rámci stejného jazyka.

- Stránka byla naposledy editována 11. 8. 2010 v 07:25.

Ekvivalence konečných automatů a regulárních gramatik.

Z Na státnice zvesela!

Regulární gramatiky popisují všechny *regulární jazyky* a v tomto smyslu (ve schopnosti popisu jazyka) jsou ekvivalentní s konečnými automaty a regulárními výrazy. Regulární gramatika ja buď pravá regulární (neterminály jsou vpravo) nebo levá regulární (neterminály jsou vlevo).

Regulární jazyk je formální jazyk (množina (i nekonečná) slov složených z omezené abecedy), který:

- může být akceptován deterministickým/nedeterministickým konečným stavovým automatem
- lze popsat regulárním výrazem
- lze ho generovat regulární gramatikou

Příkladem **ne**regulárního jazyka je $a^n b^n$, kde n > 1 (alespoň jedno *a* následované stejným počtem *b*).

Postup převodu gramatiky na konečný automat

Potřebujeme získat regulární gramatiku ve standardní formě

- *Pravá regulární gramatika* je taková, která obsahuje pouze pravidla tvaru $X \rightarrow aY$ a $X \rightarrow a$, $X \rightarrow e$ kde *X,Y* jsou neterminály, *a* je právě jeden terminál, *e* je prázdný symbol. Toho dosáhneme takto:
  - Původní gramatika typu 3 (lineární): *G = (N, T, S, P)*
  - Požadovaná regulární gramatika: *G' = (N', T, S, P')*
  - Požadovaná gramatika *G'* bude mít stejné terminální symboly a stejný počáteční stav.
  - Konstrukce přechodů *P'*:
    - do *P'* zařadíme všechna pravidla z *P* ve tvaru $X \rightarrow aY$ a $X \rightarrow e$
    - za každé pravidlo $X \rightarrow x1\ x2\ x3\ Y$ zařadíme do *P'* soustavu pravidel:
      - $X \rightarrow x1\ X1$
      - $X1 \rightarrow x2\ X2$
      - $X2 \rightarrow x3\ Y$
    - za každé pravidlo $X \rightarrow z1\ z2$, zařadíme do *P'* soustavu:
      - $X \rightarrow z1\ Z1$
      - $Z1 \rightarrow z2\ Z2$
      - $Z2 \rightarrow e$
    - místo pravidel tvaru $X \rightarrow Y$ musíme zajistit to, aby z každého stavu *X* pro který máme $X \rightarrow Y$, bylo možné odvodit všechny řetězce, které lze odvodit z *Y*.
    - *N'* vznikne obohacením *N* o všechny nově vytvořené neterminální symboly

Zkonstruujeme automat z nově vytvořené gramatiky

- stavy budou odpovídat neterminálním symbolům
- vstupy budou odpovídat terminálním symbolům
- přechodovou funkci zkonstruujeme na základě analogií
  - $X \rightarrow aY \leftrightarrow$ přechod ze stavu *X* do stavu *Y* při vstupu symbolu *a*
- počáteční stav bude odpovídat počátečnímu symbolu
- množinu koncových stavů určíme z pravidel $X \rightarrow e$

Tímto jsme získali **nedeterministický konečný automat**, který lze převést na **deterministický konečný automat** (viz otázka 28: Nedeterministický_a_deterministický_konečný_automat.)

Citováno z „http://wiki.zvesela.cz/index.php/Ekvivalence_kone%C4%8Dn%C3%BDch_automat%C5%AF_a_regul%C3%A1rn%C3%ADch_gramatik.“

- Stránka byla naposledy editována 11. 8. 2010 v 06:26.

Ekvivalence konečných automatů a regulárních gramatik. - Na státnic... http://wiki.zvesela.cz/index.php/Ekvivalence_konečných_automatů_...

Nedeterministický a deterministický konečný automat.

Z Na státnice zvesela!

Deterministický konečný automat (DKA)

- je uspořádaná pětice (S,Σ,P,s,F), kde:
  - S je konečná množina stavů.
  - Σ je konečná množina vstupních symbolů nazývaná abeceda.
  - P je tzv. přechodová funkce (též přechodová tabulka), popisující pravidla přechodů mezi stavy.
  - s je počáteční stav (s naleží S)
  - F je množina koncových stavů (F je podmnožinou S)

Nedeterministický konečný automat (NKA)

- formálně je definován jako DKA, ale obsahuje prvky nedeterminismu:

1. nejednoznačně určený počáteční stav (může jich být více)
2. nejednoznačné přechody (při přijetí stejného vstupu lze přejít do více stavů)
3. e - přechody (přechod do stavu bez přijetí vstupního symbolu)

- chování NKA lze popsat sekvencí pozic (množina stavů, ve kterých se automat může nacházet), z nichž každá jednoznačně definuje, zda je zpracovaný řetězec akceptován či zamítnut
- pozic je konečný počet
- přechody mezi pozicemi jsou jednoznačné
- to vše jsou vlastnosti DKA a proto **ke každému NKA existuje ekvivalentní DKA**

Převod NKA na DKA

Lineární gramatiku nejprve převedeme na regulární tvar (postup viz otázka [Ekvivalence konečných automatů a regulárních gramatik (http://wiki.zvesela.cz/index.php /Ekvivalence_konečných_automatů_a_regulárních_gramatik.)].

Pak zkonstruujeme nedeterministický konečný automat a z něho nakonec deterministický (jak viz dále).

Příklad

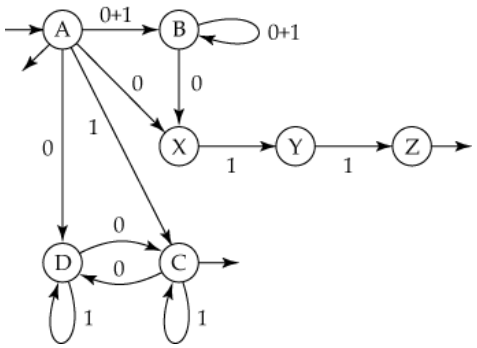
Zadaná pravá lineární gramatika:

```
A --> B | C
B --> 0B | 1B | 011
C --> 0D | 1C | e
D --> 0C | 1D
```

Pravá regulární gramatika:


```
A --> 0B | 1B | 0X | 0D | 1C | e
B --> 0B | 1B | 0X
X --> 1Y
Y --> 1Z
Z --> e
C --> 0D | 1C | e
D --> 0C | 1D
```

Nedeterministický konečný automat:



Deterministický konečný automat:

Přechodovou tabulku deterministického konečného automatu vytvoříme z přechodového diagramu nedeterministického kon. automatu takto:

- Do prvního řádku tabulky zapíšeme počáteční stav automatu a postupně zjistíme, do jakých množin stavů se nedet. automat může dostat z tohoto stavu přijmutím jednotlivých symbolů jeho vstupní abecedy.
- Z nalezených množin s více než jedním stavem vytvoříme tzv. *kompozitní* stavy det. automatu. Ty pak použijeme do přechodové tabulky det. automatu jako výstupy přechodové funkce pro počáteční stav a odpovídající vstupní symboly.
- Vzniklé kompozitní stavy (a případně i normální stavy) také využijeme v dalších řádcích přechodové tabulky a případně doplňujeme nové kompozitní stavy, do kterých se můžeme dostat z množin původních stavů každého kompozitního stavu přes vstupní symboly.
- Takto postupně vytvoříme celou přechodovou tabulku ekvivalentního deterministického automatu.
- Kompozitní stavy, zahrnující původní koncové stavy, můžeme označit také jako koncové.

```
    stav       0       1
<-->   A       BXD     BC
       BXD     BXC     BYD
<--    BC      BXD     BC
<--    BXC     BXD     BYC
       BYD     BXC     BZD
<--    BYC     BXD     BZC
<--    BZD     BXC     BD
<--    BZC     BXD     BC
       BD      BXC     BD
```

Nové stavy jsou A,BXD, BC, BXC, atd. Přechody 0,1.

Citováno z „http://wiki.zvesela.cz/index.php/Nedeterministick%C3%BD_a_deterministick

%C3%BD_kone%C4%8Dn%C3%BD_automat.“

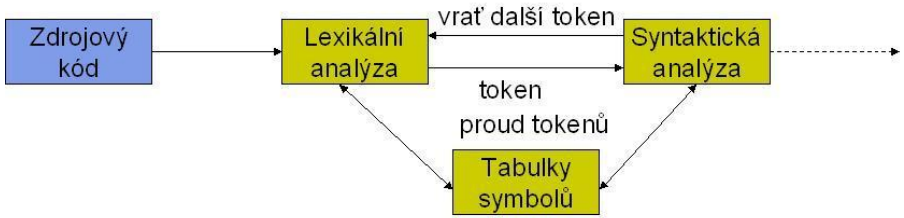
- Stránka byla naposledy editována 11. 8. 2010 v 08:54.

Lexikální analýza, princip činnosti.

Z Na státnice zvesela!

Lexikální analýza je činnost, kterou provádí lexikální analyzátor, který je vstupní a nejjednodušší částí překladače. Úkolem lexikálního analyzátoru je nalezení a rozpoznávání lexikálních symbolů (tokenů). Zdrojový program vstupuje do procesu překladu jako posloupnost znaků. Tato posloupnost se čte lineárně zleva doprava a sestavují se z ní lexikální symboly. Vynechávají se nevýznamné mezery, konce řádků a komentáře. Příkladem lexikálního elementu je číslo, identifikátor, klíčová slova (if, begin, …), =, := apod. Všechny rozpoznané lexikální symboly se ukládají do vnitřních struktur programu, tyto struktury jsou již snadno dále zpracovatelné. Vnitřní struktury většinou obsahují identifikaci o jaký typ lexikálního symbolu se jedná (např.: IDENTIFIKÁTOR, ČÍSLO, KLÍČOVÉ_SLOVO_IF, KLÍČOVÉ_SLOVO_BEGIN, …) a případný atribut. Například u lexikálního symbolu ČÍSLO by se ukládá do atributu jeho hodnota, která se vyhodnotí v průběhu lexikální analýzy.

Spolupráce lexikálního analyzátoru s syntaktickým analyzátorem vypadá takto:



Příklad rozpoznání:

```
Na vstupu: A = 10

Nalezené lexikální symboly:
               IDENTIFIKÁTOR – přidružený atribut A
               SYMBOL_PŘIŘAZENÍ
               ČÍSLO – přidružený atribut 10
```

K tomu, aby lexikální analyzátor dovedl rozpoznat ve vstupní posloupnosti znaků jednotlivé lexikální elementy, se používá konečný automat. Možné řetězcové hodnoty lexikálních elementů jsou obvykle definovány jako regulární výrazy. Při možných nejednoznačnostech, kdy je jeden symbol je prefixem jiného symbolu, se využívá pravidlo nejdelší shody.

Příslušná gramatika by mohla vypadat např. takto:

```
lexikální element = identifikátor | klíčové_slovo | číslo |speciální_symbol
speciální_symbol = '+' | '*' | '/' | '-' | '=' | '(' | ')' | '.'
identifikátor = písmeno { písmeno }
klíčové_slovo = identifikátor
písmeno = 'a' | 'b' | .... | 'A' | 'B' | .... | 'Z'
číslo = číslice { číslice }
číslice = '0' | '1' | .... | '9'
```

Nádherně popsáno na Wikipedii (http://cs.wikipedia.org/wiki/Lexik%C3%A1ln%C3%AD_anal%C3%BDza)

Citováno z „http://wiki.zvesela.cz/index.php/Lexik%C3%A1ln%C3%AD_anal%C3%BDza%2C_princip_%C4%8Dinnosti.“

- Stránka byla naposledy editována 11. 8. 2010 v 11:29.

Konstruktory lexikálních analyzátorů.

Z Na státnice zvesela!

Konstruktor lexikálního analyzátoru = generátor lex. analyzátoru. Konstruktor zpracuje vstup (sadu regulárních definic) a vygeneruje implementaci lexikálního analyzátoru.

Pravidla pro zápis regulárních výrazů:

Znaky " \ [ ] ^ - ? . * + | ( ) $ / { } % < > mají speciální význam; ostatní znaky jsou "běžné". Potřebujeme-li některý z těchto znaků použít ve významu "běžného znaku", musí být uzavřený v uvozovkách nebo musí následovat za znakem \. V následujícím seznamu x, y, z symbolizují "běžné" znaky.

LEX (LEXical analyzer generator)

Program LEX slouží k tvorbě jiných programů, které mají cosi udělat se vstupním (textovým) souborem za pomoci lexikální analýzy. Tím se myslí analýza struktur, které se dají zapsat lineárními gramatikami, konečnými automaty nebo regulárními výrazy. Typické použití LEXu je dvojí: vytvořený program pracuje samostatně, nebo slouží jako vstupní filtr pro jiný (syntaktický) analyzátor, např. bison či yacc.

Vstupem programu LEX je soubor (obvykle s koncovkou .l), který popisuje rozpoznávaná slova a akce, které se mají po jejich rozpoznání provést. Slova se popisují regulárními výrazy, akce v cílovém programovacím jazyku. Výstup LEXu je zdrojový kód hotového programu, který se potom musí běžným způsobem přeložit. Pokud tedy používáme variantu LEXu, která generuje výstup v jazyku C, musí být i akce zapsané v jazyku C. Soubory .l mají následující strukturu:

```
deklarace, definice
%%
popis slov a akcí
%%
další funkce (zapsané v cílovém jazyku)
```

Jako příklad si uvedeme program, který ve vstupním souboru nahradí všechny identifikátory slovem "IDENTIFIKATOR"

```
%%
[a-zA-Z_][0-9a-zA-Z_]*  printf("IDENTIFIKATOR");
%%
int main(void)
{
  yylex();
  return 0;
}
```

V popisu rozpoznávaných slov lze používat následující konstrukce:

```
x       znak "x"
[xy]    znak "x" nebo "y"
[x-z]   všechny znaky od "x" až k "z"
[^x]    jakýkoliv znak vyjma "x"
.       jakýkoliv znak, až na novou řádku
^x      znak "x", pokud se nachází na začátku řádky
x$      znak "x", pokud se nachází na konci řádky
x*      libovolný počet znaků "x"
x+      alespoň jeden znak "x"
x?      jeden nebo žádný znak "x"
x{m,n}  M až n výskytů znaku "x"
x|y     znak "x" nebo "y"
(x)     znak "x"
x/y     znak "x", je-li následován znakem "y"
{DEF}   doplnění definice z úvodní sekce
<y>x    znak "x", je-li splněna podmínka y
```

- **FLEX** (Fast LEXical analyzer generator) - GNU varianta LEXu

Antlr

- konstruktor lexikálních i syntaktických analyzátorů napsaný v Javě
- generuje kód analyzátoru v jazycích *C, Java, Python, C#, Objective-C*
- zápis pravidel bezkontextových gramatik v Rozvinuté Backus-Naurově formě (http://cs.wikipedia.org/wiki/Rozvinut%C3%A1_Backusova-Naurova_forma)
- příklad jednoduhé gramatiky v EBNF:

```
číslice          = "0" | číslice-bez-nuly ;
číslice-bez-nuly = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

- u každého deklarovaného neterminálního symbolu může být definován blok programového kódu, který se pak vykonává v okamžiku rozpoznání tohoto neterminálního symbolu generovaným syntaktickým analyzátorem.

- **JavaCC** - napsaný v Javě, generuje Java parsery
- **JLex, JFlex** - implementace Lexu v Jave
- **PLY** - implementace Lexu v Pythonu

- Stránka byla naposledy editována 11. 8. 2010 v 12:16.

Konstruktory lexikálních analyzátorů. - Na státnice zvesela! http://wiki.zvesela.cz/index.php/Konstruktory_lexikálních_analyzátorů.

Bezkontextové gramatiky a zásobníkové automaty.

Z Na státnice zvesela!

Obsah

- 1 Bezkontextové gramatiky
- 2 Zásobníkový automat
  - 2.1 Popis činnosti automatu
  - 2.2 Vztah bezkontextových gramatik a zásobníkových automatů
- 3 Příklady

Bezkontextové gramatiky

Bezkontextové gramatiky (BKG) jsou gramatiky typu 2 podle Chomského hierarchie.

Skládají se z pravidel $A \to \gamma$ kde A je právě jeden neterminál a γ je řetězec terminálů a neterminálů. Pravidlo $S \to e$ je povoleno, pokud se S nevyskytuje na pravé straně žádného pravidla. Jazyky generované touto gramatikou jsou **rozpoznatelné nedeterministickým zásobníkovým automatem**.

Bezkontextovou gramatiku si lze představit jako G = (N,T,P,S) , kde:

- N je množina neterminálních symbolů
- T je množina terminálních symbolů
- $S \in N$ a je to počáteční symbol
- P je množina přepisovacích pravidel ve tvaru $A \to \gamma$, kde $A \in N$ a $\gamma \in N \cup T$

Příkladem může být jazyk $L = \{0^n 1^n\}$ pro $n \geq 0$, takovýto jazyk není rozpoznatelný konečným automatem, zásobníkovým ano.

Pro tento jazyk by platilo:

$N = \{S\}$
$T = \{0,1\}$
$P = \{S \to 0S1, S \to e\}$
$S = \{S\}$

Zásobníkový automat

Formálně je zásobníkový automat definován jako uspořádaná sedmice (Q,T,G,δ, q0,z0,F), kde:

- Q je konečná množina vnitřních stavů,
- T je konečná vstupní abeceda,
- G je konečná abeceda zásobníku,
- δ je tzv. přechodová funkce, popisující pravidla činnosti automatu (jeho program), je definováno jako zobrazení $Qx(T \cup \{e\})xG^*$ do QxG^*
- q0 je počáteční stav,
- z0 popisuje symboly uložené na počátku v zásobníku,
- F je množina přijímajících stavů, $F \subseteq Q$.

Je vidět, že zásobníkový automat se v podstatě skládá z konečného automatu, který má navíc k dispozici potenciálně nekonečné množství paměti ve formě zásobníku. Obsah tohoto zásobníku ovlivňuje činnost automatu tím, že vstupuje jako jeden z parametrů do přechodové funkce.

Zásobníkový automat se od konečného automatu liší ve dvou směrech:

1. Využívá vršek zásobníku při rozhodování jaký přechod provést.
2. Může manipulovat se zásobníkem jako součást provádění přechodu.

Popis činnosti automatu

Na počátku se automat nachází v definovaném počátečním stavu a zásobník obsahuje pouze počáteční symboly. Dále v každém kroku podle aktuálního stavu, symbolů na vrcholu zásobníku a symbolu na vstupu provede přechod, při kterém může vyjmout ze zásobníku několik symbolů, vložit místo nich jiné a na vstupu přečíst další symbol. Toto se opakuje.

Po dokončení činnosti (po přečtení celého vstupu, pokud do té doby nedojde k chybě) je rozhodnuto, jestli automat vstupní řetězec přijal. K tomu mohou sloužit dvě kritéria:

- stav, ve kterém se na konci automat nachází, patří do množiny přijímajících stavů, nebo
- zásobník je na konci prázdný.

Obě definice jsou ekvivalentní, automaty na sebe lze vzájemně převádět (u druhé možnosti je možno z definice automatu zcela vypustit množinu přijímajících stavů).

Konfigurace automatu se dá popsat uspořádanou trojicí (q, w, alfa), kde q je vnitřní stav, w dosud nezpracovaná část vstupu a alfa obsah zásobníku. Na počátku práce je automat v konfiguraci (q0, w, z0).

Vztah bezkontextových gramatik a zásobníkových automatů

Zásobníkové automaty jsou ekvivalentní bezkontextovým gramatikám: pro každou bezkontextovou gramatiku existuje zásobníkový automat, který generuje (akceptuje) indentický jazyk generovaný touto gramatikou a naopak.

Pro danou BKG gramatiku W=(N, T, P, S) můžeme sestrojit zásobníkový automat P takový, že L(W)=L(P). Jsou dvě varianty:

1. Konstrukce zásobníkového automatu, který je modelem syntaktické analýzy shora dolů:
   - Q = {q} (automat má jen jeden vnitřní stav),
   - T je shodná s množinou terminálních symbolů rozpoznávané gramatiky,
   - G = N+T, tj. v zásobníku se může vyskytnout jakýkoliv symbol rozpoznávané gramatiky,
   - δ je dáno rozkladovou tabulkou,

- q0 = q, počáteční stav automatu je q, neboť automat jiné stavy nemá,
- z0 = S, tj. na počátku je v zásobníku startovací symbol gramatiky
- F = {}, což se interpretuje jako "automat akceptuje vyprázdněním zásobníku".

2. Analýza zdola nahoru je obecnější a vyžaduje trochu složitější automat:
   - Q = {q, r}, stav q je "pracovní", stav r "akceptační",
   - T je shodná s množinou terminálních symbolů rozpoznávané gramatiky,
   - G je v nejjednodušším případě rovno N+T+{#}, tj. sjednocení symbolů gramatiky a speciálního symbolu "#"; deterministický automat může mít množinu G složitější
   - d je dáno rozkladovou tabulkou,
   - q0 = q,
   - z0 = #,
   - F = {r}.

Příklady

Sestavte rozkladovou tabulku a pokuste se akceptovat řetězce "abbab" a "aaa". Gramatika G je dána:

```
S --> aAS (1)
S --> b (2)
A --> a (3)
A --> bSA (4)
```

rozkladová tabulka:

```
M|  a   b   e
--------------
S|  1   2
A|  3   4
a|  sr.
b|      sr.
e|          akc.
```

V horní řadě tabulky je počáteční symbol vstupujícího řetězce. V levém sloupci je symbol na vrcholu zásobníku. Číslo v tabulce odpovídá přepisovacímu pravidlu.

Nebudeme do konfigurace automatu zapisovat vnitřní stav q (je jen jeden). Navíc ale budeme zapisovat seznam přepisovacích pravidel, které jsme při rozpoznávání použili.

```
(abbab, S, e) :-          start
(abbab, aAS, 1) :-        na vstupu bylo a, na vrcholu S => použijeme pravidlo 1, dále jen (1)
(bbab, AS, 1) :-          srovnání
(bbab, bSAS, 14) :-       (4)
(bab, SAS, 14) :-         srovnání
(bab, bAS, 142) :-        (2)
(ab, AS, 142) :-          srovnání
(ab, aS, 1423) :-         (3)
(b, S, 1423) :-           srovnání
(b, b, 14232) :-          (2)
(e, e, 14232) :-          srovnání, akceptace (prázdný zásobník)
```

Zápis akceptace zjednodušíme: bude-li to možné, budeme dva kroky "přepsání neterminálního symbolu" a srovnání" zapisovat jako jednu činnost.


```
(aaa, S, e) :- (aaa, aAS, 1) :- (aa, AS, 1) :- (aa, aS, 13) :- (a, S, 13) :- (a, aAS, 131) :-
(e, AS, 131) :- neakceptováno
```

Pro gramatiku G sestavte rozkladovou tabulku a akceptujte řetězec "ddbcccc". G =

```
S --> dSA    (1)
S --> bAc    (2)
A --> dA     (3)
A --> c      (4)
```

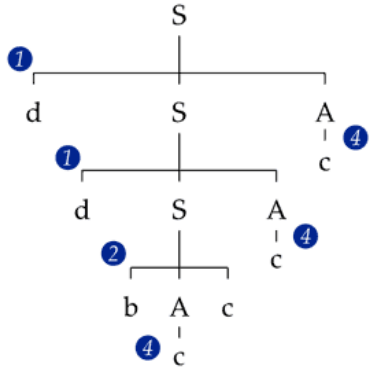
tabulka:

```
M|  b   c   d
--------------
S|  2       1
A|      4   3
```

Oproti předchozímu příkladu je tabulka zjednodušená. Chybí terminální symboly v levém sloupci a "e" v horní řadě. Tato část by byla stejná.

```
(ddbcccc, S, e) :- start
(ddbcccc, dSA, 1) :- na vrcholu zásobíku byl S, na vstupu d => podle rozkladové tabulky použijeme p
(dbcccc, SA, 1) :- srovnání
(dbcccc, dSAA, 11) :- (1) (opět použití pravidla č. 1)
(bcccc, SAA, 11) :- srovnání
(bcccc, bAcAA, 112) :- (2)
(cccc, AcAA, 112) :- srovnání
(cccc, ccAA, 1124) :- (4)
(cc, AA, 1124) :- 2x srovnání
(cc, cA, 11244) :-  (4)
(c, A, 11244) :- srovnání
(c, c, 112444) :- (4)
(e, e, 112444) :- srovnání, akceptace
```

Procházíme-li seznam použitých pravidel zleva doprava, můžeme snadno nakreslit derivační strom:



Další ukázka akceptace řetězce *abaaab* bezkontextovou gramatikou metodou shora dolů.

```
S --> aAS     (1)
S --> b       (2)
A --> bA      (3)
A --> a       (4)
```

```
(q, abaaab, S) :-        start
(q, abaaab, aAS) :-      rozklad podle (1), dále jen (1)
(q, baaab, AS) :-        srovnání terminálních symbolů, dále jen srovnání
(q, baaab, bAS) :-       (3), rozkládá se vždy nejlevější symbol
(q, aaab, AS) :-         srovnání
(q, aaab, aS) :-         (4)
(q, aab, S) :-           srovnání
(q, aab, aAS) :-         (1)
(q, ab, AS) :-           srovnání
(q, ab, aS) :-           (4)
(q, b, S) :-             srovnání
(q, b, b) :-             (2)
(q, e, e)                akceptováno
```


- Stránka byla naposledy editována 11. 8. 2010 v 12:41.

Nedeterministický syntaktický analyzátor.

Z Na státnice zvesela!

Při syntaktické analýze konstruujeme derivační strom. Podle toho, jak je konstruován derivační strom věty, rozlišujeme dvě základní metody syntaktické analýzy:

- metoda shora dolů
  - derivační strom konstruujeme od kořene k listům a zleva doprava (provádíme levou derivaci)
- metoda zdola nahoru
  - postupujeme od listů směrem ke kořeni, ale také zleva doprava (provádíme pravou derivaci)

K syntaktické analýze se využívají zásobníkové automaty (ZA), které jsou obecně nedeterministické (nepoužitelné pro SA). Pro konstrukci SA lze použít buď:

- Deterministickou simulaci nedeterministického ZA = algoritmus syntaktické analýzy s návraty.
- Zdokonalit konstrukci ZA tak, aby byl pro určitou třídu BKG deterministický (pohled do zásobníku nebo dále do vstupního řetězce).

Obecný popis nedeterminismu a determinismu

Základem **nedeterminismu** je tedy vždy problém **výběr správného pravidla**, ať už analýzou shoda dolů nebo zdola nahoru. Pokud se nemá analyzátor podle čeho rozhodnout, prostě **prochází prostor všech řešení** buď do šířky nebo do hloubky (backtracking) a hledá to správné řešení. Tato metoda je tedy značně **neefektivní**, protože v nejhorším případě může projít všechny možnosti a nenajít žádné správné řešení, tedy správnou množinu pravidel, jejichž expanzí/redukcí lze dosáhnout požadovaného výsledku.

V případě, že chceme analyzovat vstup **deterministicky**, musíme analyzátor **zdokonalit**. Ten musí mít přesnou informaci o tom, jaké pravidlo gramatiky v danou chvíli použít. Automat může využít informaci o **dosud provedené částečné derivaci** a také o **vstupu**, který ještě nebyl zpracován. Zásobníkovému automatu, který je abstraktním modelem syntaktického analyzátoru, tedy připravíme rozkladovou tabulku (lookup table), ve které bude určeno, jaké pravidlo má analyzátor použít podle vstupu a stavu zásobníku.

Metoda shora dolů

Derivační strom konstruujeme od kořene (ohodnoceného startovním symbolem) dolů k listům, zleva doprava podle levé derivace. Jedná se o zásobníkový automat LL. Počáteční konfigurace automatu se dá popsat uspořádanou trojicí (q, w, alfa), kde q je vnitřní stav, w dosud nezpracovaní část vstupu a alfa obsah zásobníku. Na počátku práce je automat v konfiguraci (q_0, w, z_0), napr. (q, abaaab, s).

Pokud jen generujeme větu v gramatice, můžeme v případě více pravidel se stejnou levou stranou náhodně vybírat. Naším úkolem však bývá spíše analýza již existující věty. Zde již náhoda nepřipadá v úvahu, protože posloupnost pravidel pro levou derivaci již nemusí být jednoznačná. Potřebujeme automat, který tuto analýzu provádí, a tento automat musí mít možnost jednoznačně vybírat mezi pravidly to správné.

Existují dva postupy (nedeterministická a deterministická):

- **analýza s návratem** (nedeterministická)
  - postupně zkoušíme vhodná pravidla. Nejdřív první, pokračujeme dále ve výpočtu, a když se ukáže, že pravidlo nevyhovuje (dostaneme se do slepé uličky), vrátíme se zpátky a vyzkoušíme druhé pravidlo atd. Tato metoda je sice účinná. ale zbytečně pomalá.
- d**eterministická analýza**
  - při výběru pravidla se řídíme dalšími informacemi. Může to být *pohled do budoucnosti*, kdy se díváme dále do vstupní posloupnosti symbolů a řídíme se tím, co později dostaneme na vstupu. Nebo například kontrolujeme obsah zásobníku (nestačí nám pouze vidět ten symbol, který ze zásobníku vyjímáme, ale i další, které jsou pod ním).

Metoda zdola nahoru

- Konstruujeme derivační strom zdola od listů nahoru ke kořeni, přičemž postupujeme zleva doprava.
- Stejně jako u první metody i zde budeme používat lineární rozklad, tentokrát pro pravou derivaci - je to proces nalezení pravého rozkladu věty (LR gramatika).
- I zde musíme rozhodovat, která pravidla chceme použít. Tentokrát však nejde o pravidla se stejnou levou stranou (pro stejný neterminál), ale rozhodujeme se mezi pravidly, která mají podobnou pravou stranu a jsou proto použitelná pro tentýž podřetězec větné formy.

Řešíme to podobně jako u předchozí metody:

- **analýza s návratem** (nedeterministicky)
  - Vybereme ve větné formě jeden podřetězec (jako první vybíráme ten, který začíná nejvíc nalevo, je co nejdelší a je shodný s pravou stranou některého pravidla), přepíšeme neterminálem na pravé straně pravidla a pokračujeme v konstrukci derivačního stromu.
  - Pokud zjistíme, že tento krok nevede k úspěchu, vyzkoušíme jiný podřetězec atd. Tato metoda je příliš časově náročná.
- **deterministická analýza** (LALR, SLR)
  - Využíváme další informace získané při překladu, např. obsah nepřečtené části vstupního kódu nebo obsah zásobníku.

Citováno z „http://wiki.zvesela.cz/index.php/Nedeterministick%C3%BD_syntaktick%C3%BD_analyz%C3%A1tor.“

- Stránka byla naposledy editována 18. 6. 2010 v 18:52.

Derivace a derivační strom, víceznačnost gramatiky.

Z Na státnice zvesela!

Gramatika G je čtveřice (N, sum, P, S), kde:

- N - množina neterminálních symbolů,
- sum - množina terminálních symbolů,
- P - konečná množina odvozovacích pravidel (sum U N)^* N (sum U N)^*,
- S - počáteční symbol tak že S náleží N.

Jazyk je vlastně množina slov, slova se skládají z písmen - tedy z terminálů. Gramatiky popisují jazyk.

P je množina pravidel, pomocí kterých lze odvodit jazyk. Jazyk S náleží N je výchozí neterminál.

Základní operací při vyvozování jazyka J je tzv. substituce, kdy za neterminály dosazujeme pravidla, která je definují. Proces postupných substitucí se nazývá derivace. Cílem derivace je vyvodit z výchozího neterminálu S platné slovo jazyka J.

DERIVACE řetězce α je posloupnost kroků odvození α pomocí přepisovacích pravidel gramatiky

S = α1 => α2 => … => α n = α

Dtto S =>* α pozn.: =>* je uzávěr relace =>

PŘÍMÁ DERIVACE α A β => α γ β ,kde A -> γ ϵ P

Derivační strom

Je grafickým vyjádřením derivace.

Říká v jakém pořadí byla jednotlivá přepisovací pravidla aplikována. Každé patro derivačního stromu odpovídá derivaci, tedy substituci neterminálního symbolu přepisovacím pravidlem.

Je orientovaný acyklický graf, který má jediný kořen, do všech ostatních uzlů vstupuje jedna hrana a má tyto vlastnosti:

- Kořen stromu je ohodnocen startovním symbolem gramatiky
- Listy jsou ohodnoceny terminálními symboly, ostatní uzly symboly neterminálními.
- Derivační strom tvoříme zleva doprava a shora dolů, proto není potřeba značit orientaci hran.

Víceznačnost gramatik

Věta generovaná gramatikou G je víceznačná, existují-li alespoň dva různé derivační stromy této věty.

Gramatiku G pak rovněž nazýváme víceznačnou.

Nutnou podmínkou jednoznačnosti gramatiky je, aby pro žádný neterminální symbol neexistovalo jak pravidlo rekurzivní zprava, tak i pravidlo rekurzivní zleva

Problém nejednoznačnosti bezkontextových jazyků je algoritmicky nerozhodnutelný.

- Stránka byla naposledy editována 4. 6. 2010 v 17:23.

Deterministická syntaktická analýza.

Z Na státnice zvesela!

Při deterministické analýze používáme některé další informace pro výběr vhodného pravidla...

viz Nedeterministický syntaktický analyzátor.

- Stránka byla naposledy editována 10. 6. 2008 v 10:14.

Rekurzivní sestup.

Z Na státnice zvesela!

Rekurzivní sestup nebo také **rekurzivní sestupný parser** postupuje shora dolů a je sestaven ze vzájemně se volajících procedur. Každá taková procedura obvykle implementuje jedno přepisovací pravidlo gramatiky.

Prediktivní parser je rekurzivně sestupný parser, který nevyžaduje zpětné kroky. Je ho možné vytvořit jen pro LL(k) gramatiky, což jsou bezkontextové gramatiky, pro které existuje kladné *k*, které umožní rekurzivně sestupnému parseru rozhodnout se, které přepisovací pravidlo použít na základě *k* dalších načtených symbolů. LL(k) gramatiky vylučují mnohoznačnost a levou rekurzi. Jakákoliv bezkontextová gramatika může být transformována na ekvivalentní nelevorekurzivní gramatiku, ale odstranění levé rekurze ne vždy vede k LL(k) gramatice. Prediktivní parser běží v lineárním čase.

Rekurzivní sestup s návratem je technika určování použitého produkčního pravidla zkoušením všech pravidel. Není limitován na LL(k) gramatiky, ale nemá zaručeno skončit, pokud gramatika není LL(k). Může vyžadovat exponenciálni čas pro svůj běh.

Princip metody rekurzivního sestupu

- každému neterminálnímu symbolu *A* odpovídá procedura *A*
- tělo procedur je dáno pravými stranami pravidel pro *A*
- pravé strany musí být rozlišitelné na základě symbolů vstupního řetězce
- jeli rozpoznána pravá strana, pak v případě neterminálního symbolu vyvolá *A* proceduru pro rozpoznaný neterminální symbol, v případě terminálního symbolu, ověří *A* jeho přítomnost ve vstupním řetězci a zajistí přečtení dalšího znaku ze vstupu
- rozpoznané pravidlo analyzátor oznámí (např. jeho číslo)
- chybnou strukturu vstupního řetězce oznámí chybovým hlášením

Sémantické zpracování

Při rekurzivním sestupu se může provádět také sémantické zpracování. Sémantické zpracování zahrnuje vyhodnocení atributů symbolů v derivačním stromu. Atributy = vlastnosti gramatických symbolů nesoucí sémantickou informaci (hodnota, adresa, typ, apod.).

Způsoby vyhodnocení:

1. procházením stromem od listů ke kořenu = **syntetizované atributy**
2. procházením stromem od rodiče k potomkovi, od staršího bratra k mladšímu = **dědičné atributy**

Je nutné doplnit procedury lex. analýzy (LA) i syntakt. ana. (SA) takto:

- LA bude předávat s přečteným vstupním symbolem i jeho atributy.
- procedury SA pro neterminály doplnit o:
  - vstupní parametry odpovídající dědičným atributům
  - výstupní parametry odpovídající syntetizovaným atributům
  - zavést lokální proměnné pro uložení atributů pravostranných symbolů
  - před vyvoláním procedury korespondujícího neterminálu z pravé strany vypočítat hodnoty jeho dědičných atributů
  - na konec procedury popisující pravou stranu pravidla zařadit příkazy vyhodnocující syntetizované atributy

Vlastnosti jazyka analyzovatelného rekurzivním sestupem

- pro metodu rekurzivního sestupu, tj. analýza shora dolů, se používají *LL* gramatiky
- jednoduchá *LL* gramatika je taková gramatika, kde levou stranu tvoří právě jeden neterminální symbol a kde každá pravá strana začíná terminálním symbolem
- navíc musí platit, že např. pro pravidla *A*->... jsou počáteční symboly různé
- obecná *LL* gramatika nemá omezení, ale musí pro ni existovat rozkladová tabulka

Principy a podmínky LL analýzy.

Z Na státnice zvesela!

Obsah

- 1 Třídy jazyků LL(k)
- 2 LL(0) gramatika
- 3 LL(1) gramatika
- 4 Mohutnosti gramatik
- 5 Typy analýzy
- 6 Analýza shora = analýza top-down
- 7 LL parsery = parsery s analýzou top-down
- 8 Syntaktická analýza LL gramatik

Třídy jazyků LL(k)

- Left to right -> vstupní text (soubor) čteme zleva doprava
- Left parse -> vytváříme levý rozklad
- při rozhodování mezi pravidly potřebujeme vidět nejvýše *k* znaků z nepřečtené části vstupu

- gramatika je typu LL(k), jestliže ji lze použít pro deterministickou syntaktickou analýzu metodou shora dolů (tj. vytváříme levý rozklad) a při rozhodování mezi pravidly potřebujeme znát nejvýše *k* symbolů ze vstupu.

- jazyk je typu LL(k), pokud je generován některou LL(k) gramatikou

LL(0) gramatika

- lze určit správné pravidlo aniž bychom předem potřebovali vidět nějaký znak na vstupu
- taková gramatika musí mít tedy pouze jedno pravidlo, jinak bychom nevěděli, které máme vybrat
- lze s ní vytvořit pouze konečnou množinu slov |L(G)| = 1
- neumožňuje rekurzi

LL(1) gramatika

- pro danou gramatiku *G* se vystačí při rozhodování o výběru pravidla pro expanzi s informací o dopředu prohlíženém řetězci délky 1 -> proto LL(1) je **gramatika silná**
- **jednoduchá LL (1) gramatika** je taková bezkontextová gramatika jestliže platí:
  - pravá strana každého pravidla začíná terminálním symbolem např. A -> aB
  - pokud mají 2 pravidla stejnou levou stranu, pak pravé strany začínají různými terminálními symboly např. A -> aB, A->bB
  - to znamená, že v každém políčku rozkladové tabulky bude právě jeden element
- **Obecná LL(1)**:
  - gramatika nemá omezení, ale musí pro ni existovat rozkladová tabulka

Mohutnosti gramatik

Typy analýzy

- shora (top-down)
- sdola (bottom-up) (vyžadují LR gramatiku, takže se netýkají této otázky)

Analýza shora = analýza top-down

- Při hledání derivace začínáme počátečním symbolem a snažíme se dostat k hledanému slovu
- LL analýza: hledáme levou derivaci, vstupní slovo analyzujeme zleva
  - Přesně určuje volbu pravidel při analýze a umožňuje jednoznačný postup při odvození
  - Gramatika, která je jednoznačná a lze ji takto analyzovat: LL gramatika
  - Využívá se zásobníkový automat
- LL(k) označení gramatiky pro LL analýzu, číslo k určuje, kolik následujících symbolů je nutné znát pro analýzu slova
- LL(1): nejpoužívanější gramatika, stačí znát jeden následující symbol
- LL(0): umožňuje jen jazyky s konečným počtem slov
- LL gramatiky s k>1 lze převést na LL gramatiky s k = 1
  - Existují přesné popisy, jak jednotlivá pravidla nahrazovat (přidávají se neterminály a pravidla se upravují, aby při analýze stačilo znát jeden další symbol)

LL parsery = parsery s analýzou top-down

LL parsery používají parsing shora dolů, zpracovávají vstup zleva doprava a konstruují nejlevější derivaci. Proto se také nazývá L (left-to-right) L(leftmost derivation). Občas se setkáváme s označením LL(k), kde k značí počet tokenů, které potřebujeme znát při rozhodování o průběhu další analýzy bez toho, aby bylo třeba používat backtracking (= prediktivní parser). Také se v této souvislosti používá pojem look-ahead. Prakticky do nedávné doby se tyto gramatiky příliš nepoužívaly, ovšem na počátku 90. let minulého století došlo ke změně přístupu.

Syntaktická analýza LL gramatik

Budeme se zabývat algoritmem syntaktické analýzy, který vytváří derivační strom analyzovaného řetězce směrem shora dolů. Základní princip syntaktické analýzy můžeme v tomto případě formulovat

takto:

Je dána bezkontextová gramatika G= (N,T,P,S) a řetězec w = a1 a2 … an, který je větou z L(G). Pak existuje levá derivace

S = γ1 ⇒ γ2 ⇒ … ⇒ γn = w.

Vzhledem k tomu, že derivace je levá, má každá větná forma γi tvar:

γi = a1 a2 … aj Ai βi,

kde a1, a2 …, aj jsou terminální symboly, Ai je neterminální symbol, βi je řetězec terminálních a neterminálních symbolů. Přitom řetězec a1a2 … aj je předponou věty w, j ≥ 0.

Podmínky LL analýzy

Předpokládejme, že A → α1 | α2 | … | αn jsou všechna pravidla v P s neterminálním symbolem A na levé straně. Pak základní problém syntaktické analýzy metodou shora dolů spočívá v nalezení toho pravidla A → αk, jehož aplikací dostaneme z větné formy γi větnou formu γi +1.

Pro výběr pravidla A → αk, je možno použít:

1. informaci o dosavadním průběhu (historii) analýzy,
2. informaci o dosud nepřečtené části vstupního řetězce (dopředu prohlíženém řetězci omezené délky).

Pokud tyto informace vždy stačí k jednoznačnému výběru pravidla A → αk, pak se gramatika G nazývá LL gramatika. Název je odvozen od toho, že při čtení vstupního řetězce zleva je vytvářen levý rozklad. Při syntaktické analýze LL gramatik jsou do zásobníku ukládány řetězce, které odpovídají levým větným formám nebo takovým jejich příponám, které vzniknou odejmutím předpony tvořené řetězcem terminálních symbolů.

Základními operacemi syntaktického analyzátoru pro LL gramatiky (LL analyzátoru) jsou:

- Expanze – neterminální symbol na vrcholu zásobníku je nahrazen pravou stranou vybraného pravidla
- Srovnání – terminální symbol na vrcholu zásobníku se ze zásobníku vyloučí, jestliže je shodný se symbolem, který byl ze vstupního řetězce přečten.
- Přijetí – vstupní řetězec je přečten a zásobník je prázdný.
- Chyba – ve všech ostatních případech.

Pokud pro danou gramatiku G vystačíme při rozhodování o výběru pravidla pro expanzi s informací o dopředu prohlíženém řetězci délky nejvýše k, pak se gramatika G nazývá *silná LL(k) gramatika*. Při analýze silných LL(k) gramatik jsou do zásobníku ukládány přímo symboly gramatiky a syntaktický analyzátor je řízen rozkladovou tabulkou.

Popis LL gramatik a LL analyzátoru (http://og2.aspweb.cz/Statnice/sw.aspx?Category=30)

Citováno z „http://wiki.zvesela.cz/index.php/Princípy_a_podm%C3%ADnky_LL_anal%C3%BDzy.“

- Stránka byla naposledy editována 14. 8. 2010 v 06:19.

Vnitřní jazyky překladačů – druhy, použití v jednotlivých fázích překladu, překlad jednoduchých jazykových konstrukcí.

Z Na státnice zvesela!

Po ukončení syntaktické a sémantické analýzy generují některé překladače explicitní intermediální reprezentaci zdrojového programu (mezikód). Intermediální reprezentaci můžeme považovat za program pro nějaký abstraktní počítač. Tato reprezentace by měla mít dvě důležité vlastnosti: měla by být jednoduchá pro vytváření a jednoduchá pro překlad do tvaru cílového programu. Intermediální kód slouží obvykle jako podklad pro optimalizaci a generování cílového kódu. Může však být také konečným produktem překladu v interpretačním překladači, který vygenerovaný mezikód přímo provádí. Intermediální reprezentace mohou mít různé formy.

Obsah

- 1 Postfixová notace
- 2 Prefixová notace
- 3 Tříadresový kód
- 4 Trojice a čtveřice
  - 4.1 Čtveřice
  - 4.2 Trojice
- 5 Příklady
  - 5.1 Tříadresový kód
  - 5.2 Čtveřice
  - 5.3 Trojice

Postfixová notace

- operátory následují ihned za operandy
- A B C * D + - => A – (B * C + D)
- efektivní zpracování pomocí zásobníku, musíme vědět prioritu operátorů

Prefixová notace

- operátory a pak operandy
- * + A B + C D => (A + B) * (C + D)

Tříadresový kód

Abstraktní forma mezikódu sestávající ze sekvence příkazů ve tvaru x := y op z, kde x, y a z jsou jména, konstanty nebo dočasné proměnné, op je nějaký operátor.

Překlad výrazu x+y*z na tříadresový kód:

t1 := y * z

t2 := x + t1

Trojice a čtveřice

Implementací tříadresového kódu jsou záznamy se třemi nebo čtyřmi poli: trojice resp. čtveřice. Následující příklady budou ukázány na výrazu: a := b * (-c) + d [b]

Čtveřice

Záznam má čtyři položky nazývané op, arg1, arg2 a res. Tříadresový příkaz ve tvaru x := y op z je reprezentován umístěním op do op, y do arg1, z do arg2 a x do res. Některé tříadresové příkazy nepotřebují všechny položky (např. x := y).

Trojice

Jestliže se chceme vyhnout generování dočasných proměnných, je možné použít formu trojic. Trojice obsahuje op, arg1 a arg2. Místo dočasných proměnných jsou indexy do pole trojic.

Příklady

Ukažme si tříadresový kód, čtveřice a trojice na příkladě výrazu:

a := b * (- c) + d [b]

Tříadresový kód

```
t1 := - c
t2 := b * t1
t3 := d [ b ]
t4 := t2 + t3
a := t4
```

Čtveřice

	op	arg1	arg2	res
(1)	uminus	c		t1
(2)	*	b	t1	t2
(3)	loadidx	d	b	t3
(4)	+	t2	t3	t4
(5)	:=	t4		a

Trojice

	op	arg1	arg2
(1)	uminus	c	
(2)	*	b	(1)
(3)	loadidx	d	b
(4)	+	(2)	(3)
(5)	:=	a	(4)

- Stránka byla naposledy editována 16. 6. 2008 v 21:04.

Tabulka symbolů – obsah, způsob manipulace při vytváření a využívání při překladu.

Z Na státnice zvesela!

Jakmile syntaktický analyzátor najde určitou konstrukci symbolů, tedy frázi, je třeba této konstrukci přiřadit význam. Součástí syntaktického analyzátoru bývá procedura (nebo více procedur či funkci), která je postupně pro každou frázi volaná a jejím úkolem je doplnit údaje do tabulky symbolů nebo do interního kódu.

Do tabulky symbolů (tabulky objektů) ukládáme postupně všechny objekty - pojmenované identifikátory (které nejsou klíčovými slovy), proměnné nebo konstanty, uživatelské datové typy, funkce, procedury, návěští apod., na které v kódu narazíme. Pojem objekt zde budeme chápat obecněji než je obvyklé v teorii programování, bude to prostě jakýkoliv identifikátor, který není klíčovým slovem a lexikální analýza ho proto odlišila od jiných identifikátorů.

Zapisujeme zde obvykle název, typ, adresu, případně počáteční hodnotu objektu, počet a typ parametrů funkce a další informace potřebné při dalším překladu, ale také při provádění programu.

Tabulka symbolů může vypadat takto:

Název	Typ	Délka	Deklarováno	Adresa	Použito
`delky`	integer array 10	40 B	A	.....	N
`I`	byte	1 B	A	.....	A
`pocet`	integer	4 B	A	.....	N
`x1`	real	6 B	A	.....	N
`z1`	*nedefinováno*	0	N	0	A

V tabulce vidíme objekty délky (pole o délce 10 prvků, prvky jsou celá čísla), I, pocet a x1, které již byly deklarovány a objekt I také použit. Objekt z1 ještě nebyl deklarován, ale už je v kódu použit. V jazyce, který umožňuje pracovat pouze s deklarovanými proměnnými, se jedná o sémantickou chybu.

U každého typu objektu potřebujeme uchovávat různé druhy informací. Například u proměnné je to název, adresa, datový typ, velikost potřebné paměti apod., u funkce název, adresa, návratový typ, počet a typ jednotlivých parametrů, příp. zda jsou volány hodnotou nebo odkazem (jestliže jsou volány odkazem, musí sémantický analyzátor navíc ošetřit, aby ve volání funkce byly jako skutečné parametry použity pouze názvy proměnných a nikoli například výrazy nebo konstantní hodnoty), u dalších typů objektů to budou opět jiné údaje. Řádky tabulky mohou být navzájem závislé (jeden uživatelský datový typ může využívat deklaraci již dříve uvedeného, popř. proměnná je typu deklarovaného dříve, . . .), nesmí se však jednat o kruhovou závislost.

Tato tabulka nám slouží k mnoha účelům. Využívá ji zejména sémantický analyzátor (kontroluje, zda proměnná použitá v kódu je deklarovaná a zda její datový typ odpovídá jejímu použití, jestli u funkce souhlasí počet a typ argumentů, atd.), používá se také u generování cílového kódu (překladač musí vědět, kolik místa v paměti má vyhradit pro jednotlivé symboly).

Při interpretaci obvykle není nutné uchovávat informaci o adrese, samotná tabulka symbolů může sloužit jako úschovna symbolů, se kterou pak neustále pracujeme.

Tabulka symbolů může být vytvářena již lexikálním analyzátorem, ten však má omezené možnosti při zjišťování některých údajů, proto je v mnoha případech vhodnější přenechat tuto práci syntaktickému nebo sémantickému analyzátoru. Často používaný postup je vytváření tabulky lexikálním analyzátorem (kdykoliv narazí na identifikátor, který není klíčovým slovem, uloží ho do tabulky) s tím, že další části překladače doplňují zbývající informace o vlastnostech uloženého identifikátoru.

Otázkou je, jak vlastně řadit jednotlivé objekty v tabulce. Důležitým kritériem je rychlost vyhledávání, protože k tabulce symbolů přistupuje zejména sémantický analyzátor velmi často. U jednodušších jazyků je možné tabulku automaticky řadit podle abecedy, u složitějších jazyků řešíme indexací, kdy zároveň s tabulkou vytváříme indexový seznam (příp. soubor), ve kterém jsou odkazy na objekty seřazené podle abecedy.

Speciální implementaci vyžaduje tabulka symbolů pro jazyk s blokovou strukturou, jako je třeba Pascal. Rozlišují se zde lokální a globální objekty a přístupnost lokálních je omezena. Každá proměnná je viditelná v tom bloku, ve kterém je deklarovaná, a také ve všech blocích vnořených. Když v určitém bloku použijeme proměnnou, hledáme informace o ní nejdřív v tom bloku, ve kterém se nacházíme. Při neúspěchu se posouváme do nadřízeného bloku a tak postupujeme, dokud ji nenajdeme. Pokud neuspějeme ani v hlavním bloku, znamená to, že byla použita proměnná, která není deklarovaná, jde o sémantickou chybu. Každý blok má svoji vlastní tabulku. S celou strukturou se pracuje jako s klasickým zásobníkem. Každá z tabulek má svou vlastní organizaci a je z ní přístupná nadřízená tabulka. „Aktivní tabulka je na vrcholu zásobníku, kde také začínáme prohledávat. Při vyhodnocení konce bloku se aktivní tabulka ze zásobníku odstraní. Po jejím odstraněni se sem přesune nadřízená tabulka. Tabulka hlavního bloku zůstává v zásobníku až do konce vyhodnocování programu, je odstraněna až jako poslední po vyhodnocení celého programu.

Tabulka symbolů

- uchovává informace o objektech
- umožňuje kontextové kontroly
- umožňuje operace

1. inicializaci informace pro standardní jména
2. vyhledání jména
3. doplnění informace ke jménu
4. přidání položky pro nové jméno
5. vypuštění položky či skupiny položek

Struktura tabulky symbolů

- s jednoduchou strukturou
- s oddělenou tabulkou identifikátorů
- s oddělenou tabulkou informací
- uspořádané do podoby zásobníku
- s blokovou strukturou

Implementace tabulky symbolů

- Vyhledávací netříděné tabulky (jen pro krátké programy)
  - prostá struktura
  - lineární seznam

- Vyhledávací setříděné tabulky
  - průběžné setřiďování
  - setřídění po zaplnění
- Frekvenčně uspořádané tabulky
- Binární vyhledávací stromy
- Tabulky s rozptýlenými položkami

Ukládání polí a struktur

Pole i struktury mají pevnou adresu začátku pole a pro přístup k jednotlivým prvkům se výsledná adresa dopočítává. Pole mohou být v paměti uložena buď po řádcích nebo po sloupcích. Tomu musí odpovídat mapovací funkce, která vypočítává relativní adresu prvků. K této adrese musí být připočtena adresa začátku pole.

Citováno z „http://wiki.zvesela.cz/index.php/Tabulka_symbol%C5%AF_%E2%80%93_obsah%2C_zp%C5%AFsob_manipulace_p%C5%99i_vytv%C3%A1%C5%99en%C3%AD_a_vyu%C5%BE%C3%ADv%C3%A1n%C3%AD_p%C5%99i_p%C5%99ekladu.“

- Stránka byla naposledy editována 16. 6. 2008 v 21:12.

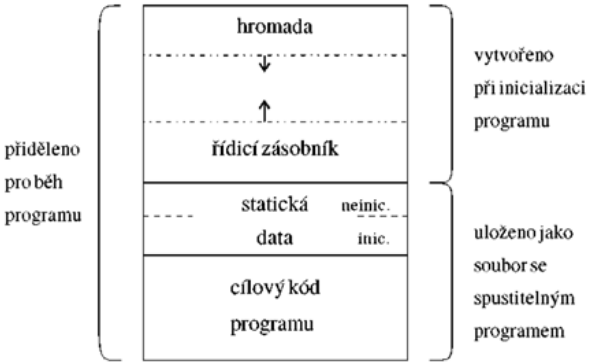
Principy přidělování paměti překladačem.

Z Na státnice zvesela!

Přeložený program dostane od operačního počítače k dispozici blok paměti, který obecně může být rozdělen na následující části:

- Vygenerovaný cílový kód
- Statická data
- Řídící zásobník
- Hromada

Velikost vygenerovaného kódu je známa již v době překladu, takže jej může překladač umístit do staticky definované oblasti, obvykle na začátek přiděleného paměťového prostoru. Rovněž velikost statických datových objektů může být známa již v době překladu a překladač je může umístit za program nebo uložit dokonce jako součást programu (to lze pouze u těch programovacích jazyků, které neumožňují rekurzivní volání procedur – Fortran). Jazyky umožňující rekurzi (Pascal, C, …) využívají pro aktivace podprogramů řídicího zásobníku, do kterého se ukládají jednotlivé aktivační záznamy. Pro účely dynamického přidělování paměti (explicitně vyžadovaného voláním příslušných funkcí nebo implicitně při přidělování paměti například pro pole s dynamickými rozměry) se používá zvláštní část paměti zvané hromada. Vzhledem k tomu, že se velikosti použité části paměti pro zásobník a hromadu v průběhu činnosti programu mohou značně měnit, je výhodné pro obě části využít opačné konce společné části paměti – viz obrázek. Nedostatek paměti se rozpozná tehdy, jestliže ukazatel konce některé oblasti překročí hodnotu ukazatele konce druhé oblasti.



Pro zmíněné datové oblasti se používají následující hlavní metody přidělování paměti:

- Statické přidělování paměti v době překladu
- Přidělování paměti na zásobníku
- Přidělování paměti z hromady

Statické přidělování paměti v době překladu

Při statickém přidělování paměti jsou všem objektům v programu přiděleny adresy již v době překladu. Při kterémkoliv volání podprogramu jsou jeho lokální proměnné vždy na stejném místě, což umožňuje

zachovávat hodnoty lokálních proměnných nezměněné mezi různými aktivacemi podprogramu. Statická alokace proměnných však klade na zdrojový jazyk určitá omezení. Údaje o velikosti a počtu všech datových objektů musejí být známy již v době překladu, rekurzivní podprogramy mají velmi omezené možnosti, neboť všechny aktivace podprogramu sdílejí tytéž proměnné, a konečně nelze vytvářet dynamické datové struktury.

Přidělování na zásobníku

Přidělování paměti pro aktivační záznamy na zásobníku se používá běžně u jazyků, které umožňují rekurzivní volání podprogramů nebo které používají staticky do sebe zanořené podprogramy. Paměť pro lokální proměnné je přidělena při aktivaci podprogramu vždy na vrcholu zásobníku a při návratu je opět uvolněna. To ale zároveň znamená, že hodnoty lokálních proměnných se mezi dvěma aktivacemi podprogramu nezachovávají.

Při implementaci přidělování paměti na zásobníku bývá jeden registr vyhrazen jako ukazatel na začátek aktivačního záznamu na vrchol zásobníku. Vzhledem k tomuto registru se pak počítají všechny adresy datových objektů, které jsou umístěny v aktivačním záznamu. Naplnění registru a přidělení nového aktivačního záznamu je součástí volací posloupnosti, obnovení stavu před voláním se provádí během návratové posloupnosti. Volací (a návratové) posloupnosti se od sebe v různých implementacích liší. Jejich činnost bývá rozdělena mezi volající a volaný program. Obvykle volající program určí adresu začátku nového aktivačního záznamu (k tomu potřebuje znát velikost záznamu vlastního), přesune do něj předávané argumenty a spustí volaný podprogram zároveň s uložením návratové adresy do určitého registru nebo na známé místo v paměti. Volaný podprogram nejprve uschová do svého aktivačního záznamu stavovou informaci (obsahy registrů, stavové slovo procesoru, návratovou adresu), inicializuje svá lokální data a pokračuje zpracováním svého těla. Při návratu opět volaný podprogram uloží hodnotu výsledku do registru nebo do paměti, obnoví uschovanou stavovou informaci a provede návrat do volajícího programu. Ten si převezme návratovou hodnotu a tím je volání podprogramu ukončeno.

Přidělování z hromady

Strategie přidělování na zásobníku je nepoužitelná, pokud mohou hodnoty lokálních proměnných přetrvávat i po ukončení aktivace, případně pokud aktivace volaného podprogramu může přežít aktivaci volajícího. V těchto případech přidělování a uvolňování aktivačních záznamů se mohou překrývat, takže nemůžeme paměť organizovat jako zásobník. Aktivační záznamy se mohou v těchto nejobecnějších situacích přidělovat z volné oblasti paměti (hromady), která se jinak používá pro dynamické datové struktury vytvářené uživatelem. Přidělené aktivační záznamy se uvolňují až tehdy, pokud se ukončí aktivace příslušného podprogramu nebo pokud už nejsou lokální data potřebná.

Při použití této strategie se pro vlastní přidělování a uvolňování paměti používají stejné techniky jako pro dynamické proměnné.

Citováno z „http://wiki.zvesela.cz/index.php/Princípy_p%C5%99id%C4%9Blov%C3%A1n%C3%AD_pam%C4%9Bti_p%C5%99eklada%C4%8Dem.“

- Stránka byla naposledy editována 2. 6. 2008 v 20:45.

Vlastnosti jazykových konstrukcí pro statický a pro dynamický způsob přidělování paměti.

Z Na státnice zvesela!

Důležitá hlediska jazykových konstrukcí:

- Dynamické typy
- Dynamické proměnné
- Rekurze
- Konstrukce pro paralelní výpočty

Podstatný je rovněž způsob:

- Omezování existence entit v programu
- Předávání parametrům fcí (hodnotou, odkazem)
- Určování přístupu k nelokálním entitám
  - na základě statického vnořování rozsahových jednotek,
  - na základě dynamického vnoření rozsahových jednotek.

Statické přidělování paměti:

- Globální proměnné
- Static proměnné
- Proměnné jazyka bez rekurze (i s blokovou strukturou) (možno staticky na zásobníku)

Předávání parametrů podprogramům

- hodnotou (C, C++, Java, C#) formální parametr podprogramu je lokální proměnnou (tohoto podrogramu) do níž se předá hodnota
- odkazem (C, C++ je-li parametrem pointer, objektové parametry Javy, C# označené ref) předá informaci o umístění skutečného parametru
- výsledkem - formální parametr je lokální proměnnou z níž se předá hodnota do skutečného parametru před návratem z podprogramu

Dynamické přidělování v zásobníku

Aktivační záznam obsahuje místo pro:

- Lokální proměnné
- Parametry
- Návratovou adresu
- Funkční hodnotu (je-li podpr. funkcí)
- Pomocné proměnné (pro mezivýsledky)
- Další informace potřebné k uspořádání aktivačních záznamů

Citováno z „http://wiki.zvesela.cz/index.php/Vlastnosti_jazykov%C3%BDch_konstrukc%C3%AD_pro_statick%C3%BD_a_pro_dynamick%C3%BD_zp%C5%AFsob_p%C5%99id

%C4%9Blov%C3%A1n%C3%AD_pam%C4%9Bti.“

- Stránka byla naposledy editována 14. 8. 2010 v 05:56.