**Системне програмне забезпечення**

**(реферат)**

1. **Поняття обчислювального процесу і ресурсу**

Поняття *“процес”* є одним основним при розгляді операційних систем. Послідовний процес (який іноді називається задачею) – це виконання окремої програми з її даними на послідовному процесорі.

Прикладом можуть бути: редагування тексту, трансляція програми, її компоновка, виконання.

І ще одне поняття є основним при розгляді ОС. Це “ресурс”.

Термін *ресурс* застосовується по відношенню до повторно використовуваних, відносно стабільних і часто невистачаючих об’єктах, які запитуються, використовуються і звільняються процесами в період їхньої активності. Іншими словами – ресурсом називається будь-який об’єкт, який може розподілятися всередині системи.

Ресурси можуть бути поділюваними, коли декілька процесів можуть їх використовувати одночасно (в один і той момент часу) або паралельно (на протязі деякого інтервалу часу процеси використовують ресурс поперемінно), а можуть бути неподільними.

Існують наступні режими організації обчислювального процесу:

пакетний режим;

режим розподілу часу (РРЧ);

режим реального часу.

Пакетний режим.

*Системи пакетної обробки* призначались для рішення задач здебільше обчислювального характеру, які не вимагають швидкого отримання результатів.

На початку роботи формується пакет завдань. Кожне завдання містить вимоги до системних ресурсів. Із даного пакету формується мультипрограмна суміш, тобто множина одночасно виконуваних задач (у відповідності до розподілу між ресурсами).

Характеризується максимальною пропускною здатністю та максимальною завантаженістю процесора та оперативної пам’яті. В цьому режимі користувач не має безпосереднього доступу до ЕОМ (саме під час виконання пакету програм).

Режим розподілу часу (РРЧ). В цьому режимі кожен користувач (а їх декілька) має безпосередній доступ до ЕОМ через свій термінал. Мета цього режиму – обслужити кожного користувача, забезпечивши йому допустимий час реакції ЕОМ на його запити.

*В системах розподілу часу* кожному користувачеві надається термінал, з якого він може вести діалог зі своєю програмою. Кожній задачі виділяється квант процесорного часу і жодна програма не займає CPU надовго. Це зручно для роботи користувача.

Кожній програмі, що готова до виконання, планується для виконання на процесорі фіксований, наперед визначений інтервал часу (інтервал мультиплексації). Програма, котра розпочне виконуватись на цьому інтервалі, може бути до кінця не виконаною в момент завершення інтервалу. Тоді її виконання примусово переривається і програма переводиться у стан очікування, що рівносильно розміщенню її у кінці черги серед готових до виконання завдань. Із початку цієї черги береться інша програма, на яку знов планується фіксований інтервал. І все повторюється знов.

*Режим реального часу*

*Системи реального часу* призначені для керування тих об’єктів, для яких існує обмежений допустимий час, в межах якого задача має бути виконана (Напр. керування супутником, науковим експериментальним обладнанням…)

Критерій ефективності – здатність витримати наперед задані інтервали часу між запуском програми та отриманням результатів).

Цей час називають реакцією системи, а відповідну властивість системи – реактивністю.

Контроль та керування відбувається в темпі, що узгоджений із швидкістю подачі даних від об’єкта керування. На обробку даних накладаються певні часові обмеження. Тому такий режим організації обчислювального процесу називають режимом реального часу і його організація надається спеціалізованій ОС.

Деякі ОС можуть суміщати властивості систем різних типів, тоді один з режимів буде називатися *фоновим.*

*Діаграма станів процесу*

Якщо розглядати не тільки звичайні ОС загального застосування, а й, наприклад, ОС реального часу, то можна сказати, що процес може знаходитися в активному та пасивному станах. В *активному стані* процес може брати участь в конкуренції за використання ресурсів обчислювальної системи, а в пасивному – він тільки відомий системі, але в конкуренції не приймає участь. В свою чергу, активний процес може бути в одному із наступних станів:

*виконання* – всі ресурси, які вимагає процес виділені. В цьому стані в кожний момент часу може знаходитися тільки один процес, якщо мова йде про однопроцесорну обчислювальну систему;

*готовності* *до виконання* – ресурси можуть бути надані, коли процес перейде стан виконання;

*блокування* або *очікування* – ресурси, які вимагаються не можуть бути надані, або не закінчена операція введення/виведення.

В звичайних ОС процес з’являється при запуску програми. ОС організує (породжує або виділяє) для нового процесу відповідний дескриптор процесу, і процес (задача) починає розвиватися (виконуватися). Тому пасивного стану не існує. В ОС реального часу ситуація інша. Звичайно при проектуванні систем реального часу заздалегідь відомий склад програм (задач) , які будуть виконуватися. Відомі і їх параметри, які необхідно враховувати при розприділенні ресурсів (напр. об’єм пам’яті, пріоритет, середня тривалість виконання, файли, які відкриваються…). Тому для них заздалегідь відводять дескриптори задач, щоби потім не витрачати дорогоцінний час на організацію дескриптора та пошук для нього необхідних ресурсів. Таким чином, в ОСРЧ багато процесів (задач) може знаходитись в стані бездії, що ми відобразили на мал., відділив цей стан від інших станів пунктиром.

За час існування процес може багато разів переходити з одного стану в інший. Це обумовлено звертаннями до операційної системи с запитами ресурсів та виконання системних функцій, які надає ОС, взаємодією з іншими процесами, появою сигналів переривання від таймера, каналів і пристроїв введення/виведення, а також інших пристроїв. Можливі переходи процесу із одного стану в інший відображені у вигляді графа на мал. Розглянемо ці переходи з одного стану в інший більш детально.

Бездіяльність

(пасивний стан)

Готовність

до

виконання

Очікування

(стан блокування)

Виконання

*Рис. 1 Граф станів процесу*

*Процес із пасивного стану може перейти в стан готовності в наступних випадках:*

по команді оператора (користувача). Має місце в тих ОС, де програма може мати статус задачі (і при цьому бути пасивною), а не просто бути виконуваним файлом і тільки на час виконання отримувати статус задачі (як в більшості сучасних ОС для ПК);

при виборі з черги планувальником (характерно для ОС, які працюють в пакетному режимі);

по виклику з іншої задачі (шляхом звернення до супервізору один процес може створювати, ініціювати, призупинити, зупинити, знищити інший процес);

по перериванню від зовнішнього ініціативного пристрою;

при настанні запланованого часу запуску програми.

Останні два способи запуску задачі, при яких процес із стану бездіяльності переходить у стан готовності, характерні для ОС реального часу.

Процес, який може виконуватися, як тільки йому буде наданий процесор, знаходиться у стані *готовності*. Вважається, що такому процесу уже виділені всі необхідні ресурси за винятком процесорного часу.

Із стану *виконання* процес може вийти по одній із наступних причин:

процес завершується, при цьому він за допомогою звернення до супервізору передає керування операційній системі і повідомляє про своє завершення. В результаті цих дій супервізор або передає його в список бездіяльних (пасивних) процесів, або знищує. В пасивний стан процес може бути переведений примусово: по команді оператора або шляхом звернення до супервізору операційної системи з іншої задачі з вимогою зупинити даний процес;

процес переводиться супервізором операційної системи в стан готовності до виконання в зв’язку з появою більш пріоритетної задачі або в зв’язку із закінченням наданого йому кванту часу;

процес блокується (переводиться в стан очікування) або внаслідок запиту операції введення/виведення (яка повинна бути виконана перед тим чим як він зможе продовжити виконання), або в силу неможливості надати йому ресурс, які необхідний йому в даний час, а також по команді оператора.

При настанні відповідної події (закінчилась операція вв/вив, звільнився необхідний ресурс і.т.д.) процес деблокується і переходить в стан готовності до виконання.

**2. Планування і диспетчеризація процесів і задач в ОС**

*Стратегії планування*

Стратегія планування визначає, які процеси ми плануємо на виконання для того, щоб досягти поставленої мети. Є велика кількість стратегій вибору процесу, але серед них найчастіше виділяють наступні:

По можливості закінчувати обчислення в тому ж порядку, в якому вони були отримані;

Надавати перевагу більш коротким процесам;

Надавати усім користувачам (процесам) однакові послуги, в тому числі однаковий час очікування.

*Дисципліни диспетчеризації*

Відома велика кількість правил (дисциплін диспетчеризації), у відповідності з якими формується список (черга) готових до виконання задач. Розрізняють два великих класи дисциплін обслуговування — *безпріоритетні і пріоритетні.* При безпріоритетному обслуговуванні вибір задачі відбувається в деякому заздалегідь установленому порядку без обліку їхньої відносної важливості і часу обслуговування. При реалізації пріоритетних дисциплін обслуговування окремим задачам дається переважне право потрапити в стан виконання.

Існуючі дисципліни диспетчеризації процесів можуть бути розбиті на два класи — витісняючі (preemtive) і невитісняючі (non-preemptive).

*Невитісняюча багатозадачність*- тобто диспетчеризація без перерозподілу процесорного часу. При такому способі активний процес виконується до тих пір, поки він самий не віддасть керування диспетчеру задач для вибору з черги іншого, готового до виконання завдання.

*Витісняюча багатозадачність* - диспетчеризація з розподілом процесорного часу. Це такий спосіб, при якому, рішення про переключення процесора з однієї задачі на виконання іншого приймається диспетчером задач, а не самим завданням.

Розглянемо коротко деякі основні (найбільш часто використовувані) дисципліни диспетчеризації.

Найпростіший у реалізації є *дисципліна FCFS* (first come — firsi served), відповідно до якої задачі обслуговуються «у порядку черги», тобто в порядку їх появи. Ті задачі, що були заблоковані в процесі роботи (потрапили в яке-небудь зі станів очікування, наприклад, через операції введення/виведення), після переходу в стан готовності ставляться в цю чергу готовності перед тими задачами, що ще не виконувалися. Іншими словами, утворяться дві черги, одна черга утвориться з нових задач, а друга черга - з тих, що раніше виконувалися, але, потрапили в стан очікування. Такий підхід дозволяє реалізувати стратегію обслуговування «по можливості закінчувати обчислення в порядку їхньої появи». Ця дисципліна обслуговування не вимагає зовнішнього втручання в хід обчислень, при ній не відбувається перерозподіл процесорного часу.

Виконані задачі

Процесор

Черга задач, які знову готові до виконання

Черга нових задач

*Дисципліна обслуговування SJN* (shortest job next) вимагає, щоб для кожного завдання була відома оцінка в потребах машинного часу. Необхідність повідомляти ОС характеристики задач, у яких описувалися би потреби в ресурсах обчислювальної системи, привела до того, що були розроблені відповідні мовні засоби. Зокрема, мова JCL (job control language) була однією з найбільш відомих. Користувачі змушені були вказувати передбачуваний час виконання, і для того, щоб вони не зловживали можливістю вказати свідомо менший час виконання (з метою одержати результати раніше від інших), ввели підрахунок реальних потреб. Диспетчер задач порівнював замовлений час і час виконання, і у випадку перевищення зазначеної оцінки в даному ресурсі ставив дане завдання не в початок, а в кінець черги.

Ще в деяких ОС у таких випадках використовувалася система штрафів, при якій у випадку перевищення замовленого машинного часу оплата обчислювальних ресурсів здійснювалася вже по інших розцінках.

Дисципліна обслуговування SJN припускає, що є тільки одна черга завдань, готових до виконання. І завдання, що у процесі свого виконання були тимчасово заблоковані (наприклад, очікували завершення операцій уведення/виведення), знову попадають у кінець черги готових до виконання нарівні з тими, що тільки надходять. Це приводить до того, що завдання, яким потрібно дуже небагато часу для свого завершення, змушені очікувати процесор нарівні з тривалими задачами, що не завжди добре.

Для усунення цього недоліку і була запропонована *дисципліна SRT* (shortest remaining time, наступне завдання вимагає менше всього часу для свого завершення).

Усі ці три дисципліни обслуговування можуть використовуватися для пакетних режимів обробки, коли користувач не змушений очікувати реакції системи, а просто здає своє завдання і через кілька годин одержує свої результати обчислень.

Для інтерактивних обчислень бажано насамперед забезпечити прийнятний час реакції системи і рівність в обслуговуванні. Для вирішення подібних проблем використовується дисципліна обслуговування, називана RR (round robin, кругова, карусельна), і пріоритетні методи обслуговування.

*Дисципліна обслуговування RR* припускає, що кожна задача одержує процесорний час порціями (*квантами часу, q).* Після закінчення кванта часу *q* задача знімається з процесора і він передається наступній задачі. Знята задача ставиться в кінець черги задач, готових до виконання. Для оптимальної роботи системи необхідно правильно вибрати закон, по якому кванти часу виділяються задачам.

Виконані задачі

Процесор

Черга задач, які знову готові до виконання

Нові задачі

**3. Розподіл переривань по рівнях пріоритету. Облік пріоритету. Дисципліни обслуговування переривань**

Переривання являють собою механізм, що дозволяє координувати паралельне функціонування окремих пристроїв обчислювальної системи і реагувати на особливі стани, що виникають при роботі процесора. Таким чином, переривання — це примусова передача керування від виконуваної програми до системи (а через неї — до відповідної програми обробки переривань), що відбуває при виникненні визначеної події.

Основна мета введення переривань — реалізація асинхронного режиму роботи і розпаралелювання роботи окремих пристроїв обчислювального комплексу.

Механізм переривань реалізується апаратно-програмними засобами і включає наступні елементи:

1. Установлення факту переривання.

2. Запам'ятовування стану перерваного процесу.

3. Керування апаратно передається підпрограмі обробки переривання.

4. Обробка переривання..

5. Відновлення інформації, що відноситься до перерваного процесу

6. Повернення в перервану програму.

На мал.1 показано, що при виникненні запиту на переривання природний хід обчислень порушується і керування передається програмі обробки переривання. При цьому засобами апаратури зберігається (як правило, за допомогою механізмів стекової пам'яті) адреса тієї команди, з якою варто продовжити виконання перерваної програми. Після виконання програми обробки переривання керування повертається перерваній раніше програмі за допомогою занесення в покажчик команд збереженої адреси команди.

Однак така схема використовується тільки в найпростіших програмних середовищах. У мультипрограмних операційних системах обробка переривань відбувається по більш складних схемах.

Підпрограма обробки переривань

Переривання

Відключення переривань, збереження контексту перерваної програми

Власне тіло програми обробки переривання

Відновлення контексту перерваної раніше програми, установка попереднього режиму роботи системи переривань

Виконувана програма

Рис.1 Обробка переривання

Отже, головні функції механізму переривань:

розпізнавання або класифікація переривань;

передача керування відповідно оброблювачу переривань;

коректне повернення до перерваної програми.

Переривання, що виникають при роботі обчислювальної системи, можна розділити на два основних класи: зовнішні (їх іноді називають асинхронними) і внутрішні (синхронні).

*Зовнішні переривання* викликаються асинхронними подіями, що відбуваються поза процесом, що переривається, наприклад:

переривання від таймера;

переривання від зовнішніх пристроїв (переривання по введенню/виведенню);

переривання по порушенню живлення;

переривання з пульта оператора обчислювальної системи;

переривання від іншого процесора чи іншої обчислювальної системи.

*Внутрішні переривання* викликаються подіями, що зв'язані з роботою процесора і є синхронними з його операціями. Прикладами є наступні запити на переривання:

при порушенні адресації (в адресній частині виконуваної команди зазначена заборонена чи неіснуюча адреса, звертання до відсутнього сегменту або сторінки при організації механізмів віртуальної пам'яті);

при наявності в поле коду операції незадіяної двійкової комбінації;

при діленні на нуль;

при переповненні або зникненні порядку;

при виявленні помилок парності, помилок у роботі різних пристроїв апаратури засобами контролю.

Нарешті, існують власне *програмні переривання.* Ці переривання відбуваються по відповідній команді переривання, тобто по цій команді процесор здійснює практично ті ж дії, що і при звичайних внутрішніх перериваннях.

Сигнали, що викликають переривання, формуються поза процесором чи у самому процесорі, можуть виникати одночасно. Вибір одного з них для обробки здійснюється на основі пріоритетів, приписаних кожному типу переривання.

Засоби контролю процесора

Системний таймер

Програмні переривання

Магнітні диски

Мережне обладнання

Термінали

Зовнішні

пристрої

Рис.2. Розподіл переривань по рівнях пріоритету

Програмне керування спеціальними регістрами маски (маскування сигналів переривання) дозволяє реалізувати різні *дисципліни обслуговування переривань*:

з *відносними пріоритетами,* тобто обслуговування не переривається навіть при наявності запитів з більш високими пріоритетами. Після закінчення обслуговування даного запиту обслуговується запит з найвищим пріоритетом. Для організації такої дисципліни необхідно в програмі обслуговування даного запиту накласти маски на всі інші сигнали переривання або просто відключити систему переривань;

з *абсолютними пріоритетами,* тобто завжди обслуговується переривання з найвищим пріоритетом. Для реалізації цього режиму необхідно на час обробки переривання замаскувати всі запити з більш низьким пріоритетом. При цьому можливе багаторівневе переривання, тобто переривання програм обробки переривань. Число рівнів переривання в цьому режимі змінюється і залежить від пріоритету запиту;

за *принципом стека,* чи, як іноді говорять, *по дисципліні LCFS* (1аst соme first served — останнім прийшов — першим обслугований), тобто запити з більш низьким пріоритетом можуть переривати обробку переривання з більш високим пріоритетом. Для цього необхідно не накладати маски ні на один сигнал переривання і не виключати систему переривань.

У багатьох операційних системах перші секції підпрограм обробки переривань виділяються в спеціальний системний програмний модуль, називаний *супервізором переривань.*

Супервізор переривань насамперед зберігає в дескрипторі поточної задачі робочі регістри процесору, що визначають контекст обчислювального процесу, що переривається. Далі він визначає ту підпрограму, що повинна виконати дії, зв'язані з обслуговуванням поточного запиту на переривання. Нарешті, перед тим як передати керування цій підпрограмі, супервізор переривань установлює необхідний режим обробки переривання. Після виконання підпрограми обробки переривання керування знову передається супервізору, цього разу уже на той модуль, що займається диспетчеризацією задач. І уже диспетчер задач, у свою чергу, відповідно до прийнятого режиму розподілу процесорного часу (між процесами, що виконуються,) відновить контекст тієї задачі, якій буде вирішено виділити процесор. Розглянута нами схема проілюстрована на мал.3

Переривання

Вибір готової до виконання задачі (на основі прийнятої дисципліни обслуговування)

Відновлення контексту перерваної раніше програми, установка попереднього режиму роботи системи переривань та передача керування цій задачі

Виконувана програма

Відключення переривань, збереження контексту перерваної програми, установка режиму системи переривань

(маскування)

Визначення адреси програмного модуля, що обслуговує запит переривання, і передача йому керування

Виконання коду підпрограми обробки переривання

Ця підпрограма вже не піклується про збереження контексту перерваного процесу

Супервізор переривань

Диспетчер задач

Рис. 3. Обробка переривання при участі супервізорів ОС

**4. Способи виділення пам’яті під новий розділ. Їх переваги і недоліки**

Загальноприйнята в даний час концепція віртуальної пам'яті з'явилася досить давно. Вона дозволила вирішити цілий ряд актуальних питань організації обчислень. Насамперед до числа таких питань відноситься забезпечення надійного функціонування мультипрограмних систем.

У будь-який момент часу комп'ютер виконує безліч процесів чи задач, кожна з яких має свій власний адресний простір. Тому необхідний механізм поділу невеликої фізичної пам'яті між різними задачами. Віртуальна пам'ять є одним зі способів реалізації такої можливості. Вона поділяє фізичну пам'ять на блоки і розподіляє їх між різними задачами. При цьому вона передбачає також деяку схему захисту, що обмежує задачу тими блоками, що їй належать.

Крім того, віртуальна пам'ять спрощує також завантаження програм, забезпечуючи механізм автоматичного переміщення програм, що дозволяє виконувати ту саму програму в довільному місці фізичної пам'яті.

З іншого боку існує поняття фізичної оперативної пам’яті, власне з якою і працює процесор, витягаючи з неї команди і дані і поміщаючи в неї результати обчислень.

*Фізична пам’ять* – це впорядкована множина комірок, що пронумеровані, тобто до кожної з них можна звернутись, вказавши її адресу. При цьому кількість комірок фізичної пам’яті обмежена і фіксована.

Системне програмне забезпечення повинно зв’язувати кожне ім’я, що вказане користувачем, з фізичною коміркою, тобто здійснювати відображення імен на фізичну пам’ять комп’ютера.

Логічне (символьне) ім’я

Простір імен програми

Система програмування

Віртуальна адреса

Операційна система

Фізична пам’ять комп’ютера

Комірка оперативної пам’яті

(фізична адреса)

*Рис.1. Пам’ять і відображення*

В загальному випадку відображення відбувається в два етапи: спочатку системою програмування, після чого операційною системою (використовуючи спеціальні модулі управління пам’яттю і відповідні засоби обчислювальної системи.) Між цими етапами звертання до пам’яті мають форму *віртуальної* або логічної адреси. При цьому можна сказати, що множина всіх допустимих значень віртуальної адреси для деякої програми визначають її *віртуальний адресний простір* або віртуальну пам’ять. Віртуальний адресний простір насамперед залежить від архітектури процесора і від системи програмування, при цьому практично не залежить від реального об’єму фізичної пам’яті комп’ютера.

Термін *віртуальна пам’ять* фактично відноситься до систем, які зберігають віртуальні адреси під час виконання. Так як друге відображення здійснюється в процесі виконання задачі, то адреси фізичних комірок можуть змінюватися.

*Просте неперервне розподіленння і*

*розподілення з перекриттям (оверлейні структури)*

Просте неперервне розприділенння - це найпростіша схема, відповідно до якої вся памя’ть умовно може бути розділена на три частини:

область, яку займає операційна система;

область, у якій розміщується задача, що виконується;

незайнята нічим (вільна) область пам'яті.

Будучи найпершою схемою, вона і сьогодні досить розповсюджена. Ця схема припускає, що ОС не підтримує мультипрограмування, тому не виникає проблеми розподілу памя’ті між декількома задачами. Вся область пам'яті при цьому - неперервна, що полегшує роботу системи програмування.

Щоб для задач відвести як можна більший обсяг пам'яті, операційна система будується таким чином, що постійно в оперативній пам'яті розташовуєся тільки найбільш потрібна її частина. Цю частину ОС називають *ядром*. Решта модулі ОС можуть бути звичайними диск-резидентними (чи транзитними), тобто завантажуватися в оперативну пам'ять тільки по необхідності, і після свого виконання знову звільняти пам'ять.

Така схема розподілу спричиняє два види втрат обчислювальних рерсів - втрата процесорного часу, тому що процесор простоює, поки задача очікує завершення операцій введення/виведення, і втрата самої оперативної пам'яті, тому що не кожна програма використовує всю пам'ять, а режим роботи в цьому випадку однопрограмний.

Якщо є необхідність створити програму, логічний (і віртуальний) адресний простір якої повиний бути більше, ніж вільна область пам'яті, чи навіть більше, ніж весь можливий обсяг оперативної пам'яті, то використовується розподіл з перекриттям (так звані *оверлейні структури).*

Цей метод розподілу припускає, що вся програма може бути розбита на частини - сегменти. Кожна оверлейна програма має одну головну частину (main) і кілька сегментів (segment), причому в пам'яті машини одночасно можуть знаходитися тільки її головна частина й один чи декілька сегментів.

Поки в оперативній пам'яті розташовуються сегменти, що виконуються, решта знаходяться в зовнішній пам'яті.

*Розподіл статичними і динамічними розділами*

Для організації мультипрограмного режиму необхідно забезпечити одночасне розташування в оперативній пам'яті декількох задач ( чи цілком чи їх частинами) Найпростіша схема розподілу пам'яті між декількома задачами припускає, що пам'ять, яка незайнята ядром ОС, може бути розбита на декілька безперервних частин (зон, розділів - partition). Розділи характеризуються ім'ям, типом, границями (як правило, указуються початок розділу і його довжина).

Розбивка пам'яті на декілька безперервних розділів може бути фіксованим(статичним), або динамічним (тобто процес виділення нового розділу пам'яті відбувається безпосередньо з появою нової задачі).

*Розділи з фіксованими границями*

Розбивка всього обсягу оперативної пам'яті на кілька розділів може відбуватися одноразово чи в міру необхідності оператором системи.

Приклад розбивки пам'яті на кілька розділів приведений на рис.2

Ядро ОС

Транзисторна область ОС

Задача А

Область, яка не використовується

Задача В

Область, яка не використовується

Задача С

Область, яка не використовується

Розділ №0

Розділ №1

Розділ №2

Розділ №3

Рис.2. Розподіл пам'яті розділами з фіксованими границями

Використання оверлейних структур дозволяє створювати великі складні програми й у той же час підтримувати коефіцієнт мультипрограмування (під коефіцієнтом мультипрограмування розуміють кількість паралельних виконуваних програм) на належному рівні. Перші мультипрограмні ОС будувалися за цією схемою.

При невеликому обсязі пам'яті і, отже, невеликій кількості розділів збільшити кількість паралельно виконуваних додатків можна за рахунок *своппінгу* (swapping). При своппінгу задача може бути цілком вивантажена на магнітний диск (переміщена в зовнішню пам'ять), а на її місце завантажується або більш привілейована, або просто готова до виконання інша задача, що знаходилася на диску в припиненому стані. При своппінгу з основної пам'яті в зовнішню (і назад) переміщається вся програма, а не її окрема частина.

Основним недоліком такого способу розподілу пам'яті є наявність часом досить великого обсягу невикористаної пам'яті (див. мал.2). Невикористана пам'ять може бути в кожному з розділів. Оскільки розділів декілька, то і невикористаних областей виходить мало, тому такі втрати стали називати *фрагментацією пам'яті.* В окремих розділах утрати пам'яті можуть бути дуже значними, однак використовувати фрагменти вільної пам'яті при такому способі розподілу не представляється можливим.

Бажання розроблювачів скоротити ці значні втрати привело їх до двох рішень:

а) виділяти розділ рівно такого обсягу, що потрібний під поточну задачу;

б) розміщати задачу не в одній безперервній області пам'яті, а в декількох областях.

*Розділи з рухливими границями*

Щоб позбутися фрагментації, можна спробувати розміщати в оперативній пам'яті задачі щільно, одну за іншою, виділяючи рівно стільки пам'яті, скільки задача вимагає. Спеціальний планувальник (диспетчер пам'яті) веде список адрес вільної оперативної пам'яті. З появою нової задачі диспетчер пам'яті переглядає цей список і виділяє для задачі розділ одним із трьох способів:

перша придатна ділянка;

найбільш придатна ділянка;

найбілш невідповідна ділянка.

У першому випадку список вільних областей упорядковується по адресах (наприклад, по зростанню адрес). Диспетчер пам'яті переглядає цей список і виділяє задачі розділ у тій області, що перша підійде по обсягу.

Спосіб «найбільш придатна» припускає, що список вільних областей впорядкований по зростанню обсягу цих фрагментів. У цьому випадку при перегляді списку для нового розділу буде використаний фрагмент вільної пам'яті, об’єм якої найбільш точно відповідає необхідному. Однак фрагмент, що залишився, виявляється настільки малим, що в ньому вже навряд чи вдасться розмістити який-небудь ще розділ.

Найефективнішим правилом є останнє, по якому для нового розділу виділяється «найбілш невідповідний» фрагмент вільної пам'яті. Для цієї дисципліни список вільних областей впорядковується по зменшенню обсягу вільного фрагмента, і оскільки цей фрагмент є найбільшим, то, швидше за все, після виділення з нього розділу пам'яті для задачі, область, що залишилася, ще зможе бути використана в подальшому.

**5. Дисципліни заміщення сегментів. Організація, переваги і недоліки**

*Сегментний спосіб організації віртуальної пам'яті*

Першим серед розривних методів розподілу пам'яті був сегментний. Для цього методу програму необхідно розбивати на частині і вже кожній такій частині виділяти фізичну пам'ять. Природним способом розбивки програми на частині є розбивка її на логічні елементи — так називані сегменти. Кожен сегмент розміщується в пам'яті як самостійна одиниця.

Логічно звертання до елементів програми в цьому випадку буде представлятися як вказівка імені сегмента і зсуву відносно початку цього сегменту. Фізично ім'я (або порядковий номер) сегменту буде відповідати деякій адресі, з якого цей сегмент починається при його розміщенні в пам’яті, і зсув повинний додаватися до цієї базової адреси.

Кожен сегмент, розташований у пам'яті, має відповідну інформаційну структуру, часто називану *дескриптором сегмента.* Саме операційна система будує для кожного процесу, що виконується, відповідну таблицю дескрипторів сегментів і при розміщенні кожного з сегментів в оперативній чи зовнішній пам'яті в дескрипторі відзначає його поточне місцезнаходження.

Отже, якщо необхідного сегмента в оперативній пам'яті немає, то виникає переривання і керування передається через диспетчер пам'яті програмі завантаження сегмента. Поки відбувається пошук сегмента в зовнішній пам'яті і завантаження його в оперативну, диспетчер пам'яті визначає придатне для сегмента місце.

Якщо вільного місця немає, то приймається рішення про вивантаження якого-небудь сегмента і його переміщення в зовнішню пам'ять.

Для рішення проблеми заміщення (визначення того сегмента, який повинний бути або переміщений у зовнішню пам'ять, або просто заміщений новим) використовуються наступні дисципліни (їх наз. дисциплінами заміщення):

правило FIFO (first in *—* first out: «перший прийшов першим і вибуває»);

правило LRU (1еаst recently used: «довше всього невикористовуваний»);

правило LFU (1еаst frequently used: «використовуваний рідше всіх інших»);

випадковий (random) вибір сегмента.

Перша й остання дисципліни є найпростішими в реалізації, але вони не враховують, наскільки часто використовується той чи інший сегмент і, отже, диспетчер пам'яті може чи вивантажити той сегмент, до якого в найближчому майбутньому буде звертання.

Алгоритм FIFO асоціює з кожним сегментом час, коли він був поміщений у пам'ять. Для заміщення вибирається найбільш старший сегмент.

Для реалізації дисциплін LRU і LFU необхідно, щоб процесор мав додаткові апаратні засоби. Мінімальні вимоги — складання списку, упорядкованого або по тривалості не використання (для дисципліни LRU), або по частоті використання (для дисципліни LFU).

*Сторінковий спосіб організації віртуальної пам'яті*

При такому способі усі фрагменти програми, на які вона розбивається (за винятком останньої її частини), виходять однаковими. Ці однакові частини називають сторінками і говорять, що пам'ять розбивається на фізичні сторінки, а програма — на віртуальні сторінки. Частина віртуальних сторінок задачі розміщається в оперативній пам'яті, а частина — у зовнішній. Місце в зовнішній пам'яті називають файлом підкачки або *сторінковим файлом* (swap-файлом)*,* тим самим підкреслюючи, що записи цього файлу — сторінки — заміщають один одного в оперативній пам'яті.

Розбивка всієї оперативної пам'яті на сторінки однакової величини приводить до того, що замість одномірного адресного простору пам'яті використовується двовимірний. Перша координата адресного простору — це номер сторінки, а друга координата — номер комірки всередині обраної сторінки (його називають індексом).

Для відображення віртуального адресного простору задачі на фізичну пам'ять, як і у випадку із сегментним способом організації, для кожної задачі необхідно мати *таблицю сторінок* для трансляції адресних просторів. Для опису кожної сторінки диспетчер пам'яті ОС заводить відповідний дескриптор, що відрізняється від дескриптора сегмента тим, що в ньому немає необхідності мати поле довжини — адже всі сторінки мають однаковий розмір.

При звертанні до віртуальної сторінки, якої не має в даний момент у оперативної пам'яті, виникає переривання і керування передається диспетчеру пам'яті, що повинний знайти вільне місце. Звичайно надається перша вільна сторінка. Якщо вільної фізичної сторінки немає, то диспетчер пам'яті по одній з вищезгаданих дисциплін заміщення (LRU, LFU, FIFO, random) визначить сторінку, що підлягає розформуванню або збереженню в зовнішній пам'яті.

Якщо обсяг фізичної пам'яті невеликий і навіть часто використовувані сторінки не вдається розмістити в оперативній пам'яті, то виникає так звана *пробуксовка —* ситуація, при якій завантаження потрібної нам сторінки викликає переміщення в зовнішню пам'ять тієї сторінки, з яким ми теж активно працюємо.

Таким чином, найважливішою перевагою сторінкового способу організації пам’яті є мінімальна фрагментація. Цей метод можна було назвати найкращим, якщо б не наступних дві обставини.

Перша – сторінкова трансляція віртуальної пам’яті вимагає суттєвих накладних витрат. Таблиці сторінок необхідно розміщати теж в пам’яті. Крім цього ці таблиці необхідно обробляти – з ними працює диспетчер пам’яті.

Друга – програми розбиваються на сторінки випадково, без обліку логічних зв’язків. Це приводить до того, що міжсторінкові переходи відбуваються частіше ніж міжсегментні і стає важко організовувати поділ програмних модулів між процесами, що виконуються.

Для того щоб уникнути другого недоліку, зберігши достоїнства сторінкового способу організації пам’яті, було запропоновано ще один спосіб – сегментно-сторінковий.

*Сегментно-сторінковий спосіб організації віртуальної пам'яті*

Як і в сегментному способі розподілу пам'яті, програма розбивається на логічно закінчені частини — сегменти — і віртуальна адреса містить вказівку на номер відповідного сегмента. Друга складова віртуальної адреси — зсув відносно початку сегмента — у свою чергу, може складатися з двох полів: віртуальної сторінки й індексу. Іншими словами, виходить, що віртуальна адреса тепер складається з трьох компонентів: сегмент, сторінка, індекс.

Цей спосіб організації віртуальної пам'яті вносить ще більшу затримку доступу до пам'яті. Необхідно спочатку обчислити адресу дескриптору сегмента і прочитати його, потім обчислити адресу елементу таблиці сторінок цього сегменту і витягти з пам'яті необхідний елемент, і вже тільки після цього можна до номера фізичної сторінки приписати номер комірки в сторінці (індекс). Затримка доступу до шуканої комірки виходить принаймні в три рази більше, ніж при простій прямій адресації.

Щоб уникнути цієї неприємності, вводиться кешування, причому кеш, як правило, будується по асоціативному принципі.

**6. Транслятори, компілятори й інтерпретатори — загальна схема роботи**

*Визначення транслятора, компілятора, інтерпретатора*

*Транслятор* – це програма, що переводить вхідну програму на вихідній (вхідній) мові в еквівалентну їй вихідну програму на результуючій (вихідній) мові.

*Компілятор —* це транслятор, що здійснює переклад вихідної програми в еквівалентну їй об'єктну програму мовою машинних команд або мовою ассемблера.

Таким чином, компілятор відрізняється від транслятора лише тим, що його результуюча програма завжди повинна бути написана мовою машинних кодів чи мовою ассемблера. Результуюча програма транслятора, у загальному випадку, може бути написана на будь-якій мові. Відповідно, усякий компілятор є транслятором, але не навпаки — не всякий транслятор буде компілятором.

Необхідність компіляторів з’явилася одночасно з появою мов програмування високого рівня.

*Інтерпретатор —* це програма, що сприймає вхідну програму вихідною мовою і виконує її.

Інтерпретатор, так само як і транслятор, аналізує текст вихідної програми. Однак він не породжує результуючої програми, а відразу ж виконує вихідну відповідно до її змісту, заданим семантикою вхідної мови.

*Етапи трансляції. Загальна схема роботи транслятора*

Таблиця ідентифікаторів

Аналіз

Лексичний аналіз

Синтаксичний аналіз

Аналіз та локалізація знайдених помилок

Семантичний аналіз

Внутрішнє представлення програми

Синтез

Підготовка до генерації коду

Генерація коду

Рис. 1. Загальна схема роботи компілятора

На мал.1 представлена загальна схема роботи компілятора. З неї видно, що в цілому процес компіляції складається з двох основних етапів — синтезу й аналізу.

На етапі аналізу виконується розпізнавання тексту вихідної програми, створення і заповнення таблиць ідентифікаторів. Результатом його роботи служить деяке внутрішнє представлення програми, зрозуміле компілятору.

На етапі синтезу на підставі внутрішнього представлення програми й інформації, що міститься в таблиці (таблицях) ідентифікаторів, породжується текст результуючої програми. Результатом цього етапу є об'єктний код.

**7. Призначення й особливості побудови таблиць ідентифікаторів**

*Призначення й особливості побудови таблиць ідентифікаторів*

Компілятор повинний мати можливість зберігати всі знайдені ідентифікатори і зв'язані з ними характеристики на протязі всього процесу компіляції, щоб мати можливість використовувати їх на різних фазах компіляції.

Для цієї мети у компіляторах використовуються спеціальні сховища даних, називані таблицями символів або *таблицями ідентифікаторів*.

Будь-яка таблиця ідентифікаторів складається з набору полів, кількість яких дорівнює числу різних ідентифікаторів, знайдених у вхідній програмі. Кожне поле містить у собі повну інформацію про даний елемент таблиці. Компілятор може працювати з однією або декількома таблицям ідентифікаторів — їхня кількість залежить від реалізації компілятора.

У таблицях ідентифікаторів може зберігатися наступна інформація:

для змінних:

ім'я змінної;

тип даних змінною;

область пам'яті, зв'язана із змінною;

для констант:

назва константи (якщо воно є);

значення константи;

тип даних константи (якщо потрібно);

для функцій:

ім'я функції;

кількість і типи формальних аргументів функції;

тип результату, що повертається;

адреса коду функції.

Як правило, кожен елемент у вхідній програмі однозначно ідентифікується своїм іменем. Тому компілятору часто доводиться виконувати пошук необхідного елемента в таблиці ідентифікаторів по його імені, у той час як процес заповнення таблиці виконуються нечасто – нові ідентифікатори описуються в програмі набагато рідше, ніж використовуються. Звідси можна зробити висновок, що таблиці ідентифікаторів повинні бути організовані таким чином, щоб компілятор мав можливість максимально швидкого пошуку потрібного йому елемента.

*Найпростіші методи побудови таблиць ідентифікаторів*

Найпростіший спосіб організації таблиці полягає в тому, щоб додавати елементи в порядку їх надходження. Тоді таблиця ідентифікаторів буде представляти невпорядкований масив інформації, кожна комірка якого буде містити дані про відповідний елементі таблиці.

Пошук потрібного елемента в таблиці буде в цьому випадку полягати в послідовному порівнянні шуканого елемента з кожним елементом таблиці, поки не буде знайдений придатний. Тоді, якщо за одиницю прийняти час, затрачуваний компілятором на порівняння двох елементів (як правило, це порівняння рядків), то для таблиці, що містить N елементів, у середньому буде виконане N/2 порівнянь.

Заповнення такої таблиці буде відбуватися елементарно просто — додаванням нового елемента в її кінець, і час, необхідний на додавання елемента (Тз), не буде залежати від числа елементів у таблиці N. Але якщо N дуже велике, то пошук зажадає значних витрат часу. Такий спосіб організації таблиць ідентифікаторів є неефективним.

Пошук може бути виконаний більш ефективно, якщо елементи таблиці впорядковані (відсортовані) в прямому чи зворотному алфавітному порядку. Ефективним методом пошуку в упорядкованому списку з N елементів є бінарний або логарифмічний пошук. Символ, який варто знайти, порівнюється з елементом (N+l)/2 всередині таблиці. Якщо цей елемент не є шуканим, то ми повинні переглянути тільки блок елементів, пронумерованих від 1 до (N+l)/2-l, чи блок елементів від (N+l)/2+1 до N у залежності від того, менше чи більше шуканий елемент від того, з яким його порівняли. Потім процес повторюється над потрібним блоком у два рази меншого розміру. Так продовжується доти, поки або елемент не буде знайдений, або алгоритм не дійде до чергового блоку, що містить один чи два елементи (з якими уже можна виконати пряме порівняння шуканого елемента).

Тому що на кожнім кроці число елементів, що можуть містити шуканий елемент, скорочується наполовину, то максимальне число порівнянь дорівнює l+log2(N).

Недоліком методу є вимога впорядкування таблиці ідентифікаторів.

Таким чином, при організації логарифмічного пошуку в таблиці ідентифікаторів ми отримуємо істотне скорочення часу пошуку потрібного елемента за рахунок збільшення часу на розміщення нового елемента в таблицю. Оскільки додавання нових елементів у таблицю ідентифікаторів відбувається істотно рідше , ніж звертання до них, то цей метод варто визнати більш ефективним, чим метод організації невпорядкованої таблиці.

**8. Призначення й особливості побудови таблиць ідентифікаторів**

*Хеш-функціею* F називається деяке відображення множини вхідних елементів R на множину цілих невід’ємних чисел Z: F(r) = n, r ∈ R, n ∈ Z.

Існують різні варіанти хеш-функцій. Одержання результату хеш-функції — «хешування» — звичайно досягається за рахунок виконання над ланцюжком символів деяких простих арифметичних і логічних операцій.

Ситуація, коли двом чи більш ідентифікаторам відповідає те саме значення функції називається *колізією.*

Природно, що хеш-функція, що допускає колізії, не може бути прямо використана для хеш-адресації в таблиці ідентифікаторів.

Очевидно, що для повного виключення колізій хеш-функція повинна бути взаємно однозначна: кожному елементу з області визначення хеш-функції повинне відповідати одне значення з її множини значень, але і кожному значенню з множини значень цієї функції повинний відповідати тільки один елемент з області її визначення. Тоді будь-яким двом довільним елементам з області визначення хеш-функції будуть завжди відповідати два різні її значення.

Для рішення проблеми колізії можна використовувати багато способів. Одним з них є метод «рехешування» (або «розміщення»). Відповідно до цього методу, якщо для елемента А адреса h(А), обчислена за допомогою хеш-функції вказує на уже зайняту комірку, то необхідно обчислити значення функції n1=h1(А) і перевірити зайнятість комірки за адресою n1*.* Якщо і вона зайнята, то обчислюється значення h2(A), і так доти, поки або не буде знайдений вільна комірка, або чергове значення hi(А) співпаде з h(А). В останньому випадку вважається, що таблиця ідентифікаторів заповнена і місця в ній більше.

*Таку таблицю ідентифікаторів можна організувати по наступному алгоритму розміщення елемента.*

*Крок 1.* Обчислити значення хеш-функції n=h(A) для нового елемента А..

*Крок 2.* Якщо комірка за адресою n порожня, то помістити в неї елемент А і завершити алгоритм, інакше i:=l і перейти до кроку 3.

*Крок 3.* Обчислити ni=hi(А). Якщо комірка за адресою ni порожня, то помістити в неї, елемент А і завершити алгоритм, інакше перейти до кроку 4.

*Крок 4.* Якщо n=ni, то повідомити про помилку і завершити алгоритм, інакше і:=і+1 і повернутися до кроку 3.

*Тоді пошук елемента А в таблиці ідентифікаторів, організованої таким чином, буде виконуватися по наступному алгоритму*.

*Крок 1.* Обчислити значення хеш-функції n = h(A) для нового елемента А.

*Крок 2.* Якщо комірка за адресою n порожня, то елемент не знайдений, алгоритм завершений, інакше порівняти ім'я елемента в комірці n з ім'ям шуканого елемента А. Якщо вони збігаються, то елемент знайдений і алгоритм завершений, інакше і:=1 і перейти до кроку 3.

*Крок 3.* Обчислити ni=hi(А). Якщо комірка за адресою ni порожня чи n=ni , то елемент не знайдений і алгоритм завершений, інакше порівняти ім'я елемента в комірці ni з ім'ям шуканого елемента А. Якщо вони збігаються, то елемент знайдений і алгоритм завершений, інакше і:=і+1 і повторити крок 3.

**9. Призначення й особливості побудови таблиць ідентифікаторів**

Неповне заповнення таблиці ідентифікаторів при застосуванні хеш-функцій веде до неефективного використання всього обсягу пам'яті, доступного компілятору. Причому обсяг невикористовуваної пам'яті буде тим вище, чим більше інформації зберігається для кожного ідентифікатора. Цього недоліку можна уникнути, якщо доповнити таблицю ідентифікаторів деякою проміжною хеш-таблицею.

Такий підхід дозволяє отримати два позитивних результатів: по-перше немає необхідності заповнювати порожніми значеннями таблицю ідентифікаторів - це можна зробити тільки для хеш-таблиці; по-друге, кожному ідентифікатору буде відповідати тільки одна комірка у таблиці ідентифікаторів ( у ній не буде порожніх невикористовуваних комірок). Порожні комірки в такому випадку будуть тільки в хеш-таблиці, і обсяг невикористовуваної пам'яті не буде залежати від обсягу інформації, збереженої для кожного ідентифікатора — для кожного значення хеш-функції буде витрачатися тільки пам'ять, необхідна для збереження одного покажчика на основну таблицю ідентифікаторів.

На основі цієї схеми можна реалізувати ще один спосіб організації таблиці ідентифікаторів за допомогою хеш-функцій, називаний «метод ланцюжків». Для методу ланцюжків у таблицю ідентифікаторів для кожного елемента додається ще одне поле, у якому може міститися посилання на будь-який елемент таблиці. Спочатку це поле завжди порожнє (нікуди не вказує). Також для цього методу необхідно мати одну спеціальну змінну, котра завжди вказує на першу вільну комірку основної таблиці ідентифікаторів (спочатку — указує на початок таблиці).

*Метод ланцюжків працює в такий спосіб по наступному алгоритму*.

*Крок 1.* В усі комірки хеш-таблиці помістити порожнє значення, таблиця ідентифікаторів не повинна містити ні одної комірки, змінна *FreePtr* (покажчик першої вільної комірки) указує на початок таблиці ідентифікаторів; і:=1.

*Крок 2.* Обчислити значення хеш-функції ni; для нового елемента Аi;. Якщо комірка хеш-таблиці за адресою ni порожня, то помістити в неї значення змінної *FreePtr* і перейти до кроку 5; інакше — перейти до кроку 3.

*Крок 3.* Покласти j:=l, вибрати з хеш-таблиці адреса комірки таблиці ідентифікаторів mj і перейти до кроку 4.

*Крок 4.* Для комірки таблиці ідентифікаторів за адресою mj перевірити значння поля посилання. Якщо воно порожнє, то записати в нього адресу з перемінної *FreePtr* і перейти до кроку 5; інакше j:=j+l, вибрати з поля посилання адресу mj і повторити крок 4.

*Крок* 5. Додати в таблицю ідентифікаторів нову комірка, записати в неї інформацію для елемента Aі (поле посилання повинне бути порожнім), у змінну *FreePtr* помістити адресу за кінцем доданої комірки. Якщо більше немає ідентифікаторів, які треба розмістити в таблиці, то виконання алгоритму закінчене, інакше і:=і+1 і перейти до кроку 2.

*Пошук елемента в таблиці ідентифікаторів, організованої в такий спосіб буде виконуватися по наступному алгоритму.*

*Крок 1.* Обчислити значення хеш-функції n для шуканого елемента А. Якщо комірка хеш-таблиці за адресою n порожня, то елемент не знайдений і алгоритм завершений, інакше покласти j:=l, вибрати з хеш-таблиці адресу комірки таблиці ідентифікаторів mj=n.

*Крок 2.* Порівняти ім'я елемента в комірки таблиці ідентифікаторів за адресою mj з ім'ям елемента А. Якщо вони співпадають, то шуканий елемент знайдений і алгоритм завершений, інакше - перейти до кроку 3.

*Крок 3.* Перевірити значення поля посилання в комірці таблиці ідентифікаторів за адресою mj . Якщо воно порожнє, то шуканий елемент не знайдений і алгоритм завершений ; інакше j:=j+l, вибрати з поля посилання адреса mj і перейти до кроку 2.

При такій організації таблиць ідентифікаторів у випадку виникненні колізії алгоритм розміщає елементи в комірках таблиці, зв'язуючи їх один з одним послідовно через поле посилання. При цьому елементи не можуть попадати в комірки з адресами, що потім будуть співпадати зі значеннями хеш-функції. Таким чином, додаткові колізії не виникають. У підсумку, у таблиці виникають своєрідні ланцюжки зв'язаних елементів.

**10. Призначення й особливості побудови таблиць ідентифікаторів. Комбіновані способи побудови таблиць ідентифікаторів**

У реальних компіляторах практично завжди так чи інакше використовується хеш-адресація. Звичайно при розробці хеш-функції творці компілятора прагнуть звести до мінімуму кількість виникаючих колізій не на всій множині можливих ідентифікаторів, а на тих їхніх варіантах, що найбільше часто зустрічаються у вхідних програмах. Звичайно, узяти до уваги всі припустимі вхідні програми неможливо. Найчастіше виконується статистична обробка імен ідентифікаторів, що зустрічаються, на деякій множині типових вихідних програм, а також приймаються в увагу угоди про вибір імен ідентифікаторів, загальноприйняті для вхідної мови. Гарна хеш-функція — це крок до значного прискорення роботи компілятора, оскільки звертання до таблиць ідентифікаторів виконуються багаторазово на різних фазах компіляції.

Те, який конкретно метод застосовується в компіляторі для організації таблиць ідентифікаторів, залежить від реалізації компілятора. Той самий компілятор може мати навіть декілька різних таблиць ідентифікаторів, організованих на основі різних методів.

Як правило, застосовуються комбіновані методи. У цьому випадку, як і для методу ланцюжків, у таблиці ідентифікаторів організується спеціальне додаткове поле посилання. Але на відміну від методу ланцюжків воно має трохи інше значення. При відсутності колізій для вибірки інформації з таблиці використовується хеш-функція, поле посилання залишається порожнім. Якщо ж виникає колізія, то через поле посилання організується пошук ідентифікаторів, для яких значення хеш-функції збігаються по одному з розглянутих вище методів: неупорядкований список, упорядкований список або бінарне дерево. При добре побудованій хеш-функції колізії будуть виникати рідко, тому кількість ідентифікаторів, для яких значення хеш-функції збіглися, буде не настільки велике. Тоді і час пошуку одного серед них буде незначним (у принципі при високій якості хеш-функції підійде навіть перебір неупорядкованому списку).

Такий підхід має переваги в порівнянні з методом ланцюжків: для збереження ідентифікаторів зі співпадаючими значеннями хеш-функції використовуються області пам'яті, що не перетинаються з основною таблицею ідентифікаторів, а виходить, їхнє розміщення не приведе до виникнення додаткових колізій. Недоліком методу є необхідність роботи з динамічно поділюваними областями пам'яті. Ефективність такого методу, очевидно, у першу чергу залежить від якості застосовуваної хеш-функції, а в другу - від методу організації додаткових сховищ даних.

Хеш-адресація — це метод, що застосовується не тільки для організації таблиць ідентифікаторів у компіляторах. Даний метод знайшов своє застосування й в операційних системах, і в системах керування базами даних.

**11. Хеш-функція і хеш-адресація**

*Хеш-функціею* F називається деяке відображення множини вхідних елементів R на множину цілих невід’ємних чисел Z: F(r) = n, r ∈ R, n ∈ Z. Сам термін «Хеш-функція» походить від англійського терміна «hash function» (hash — «заважати», «змішувати», «плутати»).

Множина припустимих вхідних елементів R називається областю визначення хеш-функції. Множиною значень хеш-функції F називається підмножина М з множини цілих невід’ємних чисел Z: M ⊆ Z (M є підмножиною Z, М вкючене в Z), що містить усі можливі значення, які повертаються функцією F: ∀r ∈ R: F(r) ∈М (Для всіх r, які належ R; F(r) належ M. )

Процес відображення області визначення хеш-функції на безліч значень називається «хешуванням». При роботі з таблицею ідентифікаторів хеш-функція повинна виконувати відображення імен ідентифікаторів на множину цілих невід’ємних чисел. Областю визначення хеш-функції буде множина усіх можливих імен ідентифікаторів.

Хеш-адресація полягає у використанні значення, що повертається хеш-функцією, як адресу комірки з деякого масиву даних . Тоді розмір масиву даних повинний відповідати області значень використовуваної хеш-функціи. Отже, у реальному компіляторі область значень хеш-функціи ніяк не повинна перевищувати розмір доступного адресного простору комп'ютера.

Метод організації таблиць ідентифікаторів, заснований на використанні хеш-адресації, полягає в розміщенні кожного елемента таблиці в комірку, адресу якого повертає хеш-функція, обчислена для цього елемента. Тоді в ідеальному випадку для розміщення будь-якого елемента в таблиці ідентифікаторів досить тільки обчислити його хеш-функцію і звернутися до потрібної комірки масиву даних. Для пошуку елемента в таблиці необхідно обчислити хеш-функцію для шуканого елемента і перевірити, чи не є задана нею комірка масиву порожньою (якщо вона не порожня — елемент знайдений, якщо порожня — не знайдений).

Цей метод дуже ефективний — як час розміщення елемента в таблиці, так і час його пошуку визначаються тільки часом, затрачуваним на обчислення хеш-функції, що у загальному випадку незрівнянно менше часу, необхідного на багаторазові порівняння елементів таблиці.

Метод має два очевидних недоліки. Перший з них – неефективне використання обсягу пам'яті під таблицю ідентифікаторів: розмір масиву для її збереження повинний відповідати області значень хеш-функції, у той час як реально збережених у таблиці ідентифікаторів може бути істотно менше. Другий недолік – необхідність відповідного розумного вибору хеш-функції.

**12. Способи внутрішнього представлення програм**

Усі внутрішні представлення програми звичайно містять у собі дві принципово різні речі — оператори й операнди. Розходження між формами внутрішнього представлення полягають лише в тому, як оператори й операнди з'єднуються між собою. Також оператори й операнди повинні відрізнятися один від іншого, якщо вони зустрічаються в будь-якому порядку. За розрізнення операндів і операторів, як уже було сказано вище, відповідає розроблювач компілятора, що керується семантикою вхідної мови.

Відомі наступні форми внутрішнього представлення програм:

зв'язані облікові структури, що представляють синтаксичні дерева;

багатоадресний код з явно іменованим результатом (тетради);

багатоадресний код з неявно іменованим результатом (тріади);

обернений (постфиксна) польський запис операцій;

асемблерний код або машинні команди.

Синтаксичні дерева

Синтаксичні дерева – це структура, що представляє собою результат роботи синтаксичного аналізатора. Вона відображає синтаксис конструкцій вхідної мови і явно містить у собі повний взаємозв'язок операцій. Очевидно також, що синтаксичні дерева — це машинно-незалежна форма внутрішнього представлення програми.

Недолік синтаксичних дерев полягає в тому, що вони являють собою складні зв'язні структури, а тому не можуть бути тривіальним чином перетворені в лінійну послідовність команд результуючої програми. Проте вони зручні при роботі з внутрішнім представленням програми на тих етапах, коли немає необхідності безпосередньо звертатися до команд результуючої програми.

Синтаксичні дерева можуть бути перетворені в інші форми внутрішнього представлення програми, що представляють собою лінійні списки, з урахуванням семантики вхідної мови. Алгоритми такого роду перетворень розглянуті далі. Ці перетворення виконуються на основі принципів СК-компіляції.

*Багатоадресний код з явно іменованим результатом (тетради)*

Тетради являють собою запис операцій у формі з чотирьох складових: операції, двох операндів і результату операції. Наприклад, тетради можуть виглядати так: <операція>(<операнд1>,<операнд2>,<результат>).

Тетради являють собою лінійну послідовність команд. При обчисленні виразу, записаного у формі тетрад, вони обчислюються одна за іншою послідовно. Кожна тетрада в послідовності обчислюється так: операція, задана тетрадою, виконується над операндами і результат її виконання міститься в змінній, заданій результатом тетради. Якщо якийсь з операндів (чи оба операнда) у тетраді відсутні (наприклад, якщо тетрада являє собою унарну операцію), то він може бути опущений чи замінений порожнім операндом (у залежності від прийнятої форми запису і її реалізації).

Тетради являють собою лінійну послідовність, а тому для них нескладно написати тривіальний алгоритм, що буде перетворювати послідовність тетрад у послідовність команд результуючої грами (або послідовність команд асемблера). У цьому їхня перевага перед синтаксичними деревами. На відміну від команд асемблера тетради не залежать від архітектури обчислювальної системи, на яку орієнтована результуюча програма. Тому вони являють собою машинно-незалежну форму внутрішнього представлення програми.

Тетради вимагають більше пам'яті для свого представлення, ніж тріади, вони також не відображають явний взаємозв'язок операцій між собою. Крім того, є складності з перетворенням тетрад у машинний код, тому що вони погано відображаються в команди асемблера і машинні коди, оскільки в наборах більшості сучасних комп'ютерів рідко зустрічаються операції з трьома операндами.

*Багатоадресний код з неявно іменованим результатом (тріади)*

Тріади являють собою запис у формі трьох складових:

<операція>(<операнд1>;<операнд2>)

Їх особливістю є те, що один або обидва операнди можуть бути посиланнями на іншу тріаду. Це в тому випадку, якщо в якості операнда даної тріади виступає результат виконання іншої тріади. Тому тріади при записі нумерують послідовно для зручності посилань.

Кожна тріада обчислюється таким чином: операція, яка задана тріадою, виконується над операндами, якщо в якості одного із операндів або двох є посилання на іншу тріаду, то береться результат обчислення тієї тріади. Результат обчислення кожної тріади потрібно зберігати в тимчасовій пам’яті, так як він може знадобитися наступним тріадам. Якщо один операнд відсутній, він може бути упущений.

Переваги:

легке написання алгоритму;

легко перевести в асемблер ний код.

Недолік: необхідний алгоритм для зберігання в пам’яті проміжного результату.

Тріади є машинно незалежні, вимагають менше пам’яті ніж тетради.

*Обернений польський запис*

Перевага: ефективний для обчислення математичних виразів.

Недоліки:

необхідно використовувати стек

важко робити оптимізацію

Нехай задано арифметичний вираз виду: (A+B)\*(C+D)-E

Представимо цей вираз у вигляді польського запису: AB+CD+\*E-

Обернений польський запис володіє властивостями, які перетворюють його в ідеальну проміжну мову при трансляції:

1. обчислення виразу може проводитися шляхом одноразового перегляду, що зручно для генерації коду

2. отримання польського запису просто здійснити на основі алгоритму DX3.

**13. Визначення формальної мови і граматики**

*Формальні мови* - це математичний апарат, що дозволяє математично грамотно створити мови програмування і писати компілятори для них.

Формальну мову можна задати як послідовність слів. Слово – це послідовність символів. Тоді навіть програму можна вважати просто словом.

Словами даної мови може бути не довільний набір символів, а лексично і синтаксично правильно побудований. Для того, щоб задати граматику, треба задати множини термінальних і не термінальних символів.

*Термінальні –* це символи, які використовуються в мові, а проміжні або *нетермінальні* – це символи, які використовуються для створення слів мови. Створюються слова за граматичними правилами. Застосування правила полягає в заміні в перетворюваному рядку якоїсь послідовності символів, що співпадає з лівою або правою частиною правила.

Компілятор, отримавши на вхід програму, робить зворотну роботу. Він згортає за граматичними правилами від правої до лівої частини початкові символи.

Кінцева множина символів, яка є неподільною, називається словником або *алфавітом*, а символи, що входять в множину – буквами алфавіту. Послідовність букв алфавіту називається словом або *ланцюжком* цього алфавіту, число букв, що входить у слово, називається його довжиною.

Якщо задано алфавіт А, то А\* - це множина всіх ланцюжків, які можна побудувати з алфавіту А. $ - порожній рядок (рядок, що не містить жодної букви) також входить в А. Для позначення всіх ланцюжків алфавіту А, що не містять порожнього використовується А+.

*Формальною граматикою* Г називається сукупність таких об’єктів:

Г={VT,VN,<I>,R},

Де VT – термінальний алфавіт (словник). З букв цього алфавіту будуються ланцюжки, які породжуються граматикою.

VN – нетермінальний (допоміжний) алфавіт. Його букви використовуються при побудові ланцюжків, вони можуть входити в проміжні ланцюжки, але не можуть входити в результат побудови.

<I> - початковий символ.

R – множина правил виведення.

Множина кінцевих ланцюжків термінального алфавіту VT граматики Г, виведених з початкового символу <I> називають *мовою*, яка породжена граматикою Г і позначається L(Г).

Якщо правило виведення граматики мінить один нетермінальний символ, як в лівій, так і в правій частині, то таке правило називається *рекурсивним*.

*Типи формальних граматик*

Виділяють 4 типи формальних граматик. Ці граматики визначаються шляхом накладання обмежень на правила граматики.

Граматика типу 0 – граматика загального вигляду, немає обмежень на правила породження.

Граматика типу 1 – контекстно залежна.

Правило: χ1<A>χ2→ χ1ωχ2.

Ланцюжки χ1 і χ2 залишаються незмінними при застосуванні правил, тому їх називають *контекстом*, а граматику – *контекстно залежною.*

Граматика типу 2 – контекстно вільна.

Правило: <A>→α, де .



Ці правила слідують із правил граматики типу 1 за умови χ1 = χ2 = $.

Граматика типу 3 – автоматна.

Правила виведення: <A>→a або <A>→a<B> або <A>→<B>a, де

.



*Способи задання схем граматик*

Схема граматики містить правила виведення, які визначають синтаксис мови або всі ланцюжки породженої мови. Для задання правил використовують різні форми опису:

символічна

форма Бекуса-Наура (ФБН)

ітераційна

синтаксичні діаграми

Символьна мова передбачає використання елементів нетермінального словника і стрілки, як роздільника правої і лівої частини. Але при описі конкретних мов програмування доводиться вводити велику кількість не термінальних символів і символьна форма запису втрачає свою наочність.