А. С. СЕМЕНОВ

ПРОЕКТИРОВАНИЕ СЕТЕВЫХ ОПЕРАЦИОННЫХ СИСТЕМ

Практический курс

Москва
«Вузовская книга»
2008

УДК 681.3.07
ББК 32.81
С30

Р е ц е н з е н т ы:

кафедра автоматизации систем вычислительных комплексов факультета вычислительной математики и кибернетики МГУ;
д-р техн. наук, проф. *Е. Е. Ковшов*

Семенов А. С.

С30 Проектирование сетевых операционных систем: практический курс. — М.: Вузовская книга, 2008. — 224 с.: ил.

ISBN 978-5-9502-0370-1

Излагается внутреннее устройство и принципы проектирования операционных систем на основе категорий. Большое внимание уделяется архитектурам построения и технологии поэтапной разработки операционных систем с применением контура (выполняющегося прототипа объектно-ориентированных операционных систем). Практические работы упорядочены в соответствии с этапами разработки. Показывается на примерах метод создания и реализации категорий операционных систем на объектно-ориентированном языке. Приводятся демонстрационные примеры и перечень практических заданий для проектирования сетевых операционных систем.

Пособие будет полезно как студентам, изучающим соответствующий курс, так и профессионалам, интересующимся вопросами построения объектно-ориентированных операционных систем и систем управления ресурсами.

УДК 681.3.07
ББК 32.81

© Семенов А. С., 2008
© ЗАО «Издательское предприятие «Вузовская книга», 2008

ISBN 978-5-9502-0370-1

ВВЕДЕНИЕ

Высококвалифицированный специалист в области информационных технологий должен владеть несколькими объектно-ориентированными языками программирования: C++, Java, C#, CLOS (Common Lisp Object System), а также языками моделирования UML (Unified modeling language). Для того чтобы упростить изучение, требующее значительных усилий, предлагается изучить семантику объектно-ориентированных языков программирования в виде универсального объектно-ориентированного языка, основу которого составляют категории. Такой подход помогает сравнительно быстро понять, ради чего строится язык как система программирования и что лежит в его основе.

Совершенствование и распространение все новых компьютерных архитектур отражают их ориентацию на новые подходы к программированию. Парадигмы программирования эволюционируют, и современные архитектуры объектно-ориентированных языков программирования представляют собой интеграцию нескольких парадигм. Целесообразно понять, как идет развитие парадигм и что нового они вносят в решение проблем программирования. Анализ проблемы в терминах категорий позволяет перейти к компонентам программ, готовым к применению, что значительно сокращает время на разработку.

При проектировании программного обеспечения (ПО) на основе объектно-ориентированного подхода возникает целый ряд проблем, связанных с принятием решения о выборе отношений и взаимодействий между объектами. Принятые решения накладывают ограничения на правила взаимодействия объектов в разрабатываемой системе, влияют на архитектуру создаваемой системы и на возможность повторного использования разработанных компонент. Представление отношений и взаимодействий между объектами в виде категорий позволяет решить возникающие проблемы. В пособии дается классификация категорий, в которой рассматриваются отношения и взаимодействия между объектами.

В первой главе дается анализ архитектур парадигм программирования. Рассматриваются механизмы абстракции, гранулярности (объем программного кода) с точки зрения повторного использования кода. Проводится анализ парадигм: структурной, объектно-ориентированной. Показывается, как с целью увеличения повторного использования

кода объектно-ориентированная парадигма расширяется парадигмами: образца (pattern), контура (framework), компоненты, сервисами, агентами и аспектами. Обсуждается теория сложного документа, модель контура и приводятся примеры.

Остальные главы посвящены практическому применению контура для операционных систем и применению категорий как конструкций фундаментальных образцов объектно-ориентированного подхода.

Выполнение лабораторных работ и курсовых проектов по проектированию и реализации ОС на основе категорий направлено на освоение методов построения и проектирования операционных систем с применением объектно-ориентированных языков программирования. В пособии приводятся примеры разработки категорий операционных систем и их реализация на языке программирования C++.

Отличительной особенностью пособия является его практическая направленность на проектирование и реализацию методов управления сетевых ОС. Для облегчения процесса усвоения в качестве учебного материала используется контур (англ. framework, переводится как контур) — выполняемая программа, ориентированная на проектирование и реализацию сетевых ОС. Контур формализует концептуальные знания об архитектуре ОС и методах реализации. Код программы приведен в конце пособия в приложении.

Пособие подразделяется на темы, раскрывающие определенные методы ОС. Каждая тема содержит две лабораторные работы, при проведении которых используется контур. Порядок следования лабораторных работ организован таким образом, чтобы в результате их выполнения сложилось целостное представление о процессе разработки, архитектуре и методах сетевых ОС. Первая лабораторная работа посвящена изучению архитектуры контура и его функциональных возможностей.

Основной задачей лабораторных работ является изучение и практическое применение методов проектирования ОС и объектно-ориентированного языка программирования C++.

Курсовой проект позволяет систематизировать и углубить знания, полученные при проведении лабораторных работ. Основу курсового проекта составляют категории, разработанные в процессе выполнения лабораторных работ и их реализация с применением контура. С целью достижения поставленной задачи для лабораторных работ и курсового проекта выдается единая схема аппаратных и программных компонент системы. Эта же схема служит тестом для проверки правильности выполнения сетевой ОС и выданного задания.

ГЛАВА 1

ЭВОЛЮЦИЯ ПАРАДИГМ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Эволюция парадигм языков программирования обусловлена поиском механизмов повторного использования программного кода.

1.1. Структурная и объектно-ориентированная парадигмы

Парадигма определяет возможные типы концепций, реализуемые в условиях проблемного решения. Повторная используемость кода является следствием применения парадигм, обеспечивая обратную связь, которая заставляет ее изменяться в сторону увеличения повторяемости кода.

Повторное использование кода программ — один из критериев создания высоко-технологичного программного обеспечения, позволяющий сократить время разработки программной системы и тем самым уменьшить ее стоимость, а также сопровождение и адаптацию к новым требованиям.

Абстракция фокусируется на подобии, выбирает существенные характеристики, имеющие отношение к общим чертам, определяет единственное представление сущности, обладающей этими характеристиками, которые имеют отношение к определению каждого элемента во множестве.

Абстрагирование — выделение существенных особенностей объекта, отличающих его от всех других объектов. Механизмы абстракции позволяют реализовать повторную используемость кода программ.

Гранулярность фокусируется на размере программного кода, определяет низкую зависимость и высокую плотность программного кода. Таким образом, чем выше гранулярность, тем больше программный код, который может быть повторно использован.

Механизмы абстрагирования присутствуют во многих парадигмах языков программирования. Взаиморазвитие механизмов определяется в поступательном движении таким образом, что каждая следующая парадигма может использовать механизмы предыдущей. Так объектно-ориентированная парадигма расширяет структурную. В свою очередь, парадигмы на основе компоненты, образца, контура расширяют объектно-ориентированную с целью увеличения повторного использования уже имеющихся решений.

Структурная парадигма характеризуется программированием без оператора goto, созданием библиотек повторно используемых процедур. Механизмы абстракции: процедуры, типы, а в реляционных базах данных — отношения. Механизмы гранулярности: модуль, пакет, библиотека.

Первая абстракция в языках программирования — процедура, реализуемая в виде подпрограммы, прямо вытекала из традиционного взгляда на программные средства. Программы имели относительно простую структуру, состоящую из глобальных данных и подпрограмм. В процессе разработки программ была возможность логического разделения различных данных, но механизмы языков его не поддерживали. Ошибка имела серьезные последствия, так как язык не гарантировал целостность данных в процессе внесения значений, что нарушало надежность системы.

Применение процедур решило ряд противоречий; усилило управление алгоритмическими абстракциями, но не решило задачи использования многообразия данных.

Сравнительно новые принципы были заложены в структурном подходе: структурная иерархия, модульность, параллелизм, типизация. Дадим их определения.

Иерархия — упорядоченная система абстракций. Структурная иерархия — отражает взаимосвязи типа «часть — целое» и определяет отношение агрегирования между объектами.

Модульность — уменьшение сложности программы за счет ее разделения на фрагменты.

Параллелизм — одновременное функционирование экземпляров объектов.

Типизация — ограничения на выполнение операций, препятствующие выполнению операций между различными типами.

В совокупности эти принципы позволили вести разработку сложных проектов, которые предшествующими методами решались с трудом.

В языках программирования появились механизмы, поддерживающие стадию передачи параметров подпрограмм, и были заложены принципы структурного проектирования, отразившиеся в создании механизмов вложения подпрограмм, включая ограничения в области действия компонентов. В структурной парадигме механизм абстракции — процедура или функция — представляется следующем образом:

$y = f(x_1, x_2, ..., x_n)$ — математическое определение функции с несколькими аргументами;

y = f(T_1 x_1, T_2 x_2, ..., T_2 x_n) — определение функции с несколькими аргументами, разных типов;

y f(T_1 x_1, T_2 x_2, ..., T_2 x_n) определение функции с несколькими аргументами, разных типов в C++, убран знак =;

void f (int a) {} — определение функции (процедуры), не возвращающей параметров;

int f (int a) {return a + 1;} — определение функции, возвращающей параметры.

В C++ несколько параметров может быть возвращено по адресу переменных. Для этого в качестве аргументов должны быть заданы адреса.

Метод структурного проектирования обеспечивал создание больших систем на основе подпрограмм. Создавались библиотеки — концепция для повторного использования процедур. Библиотеки переносились с компьютера на компьютер.

Процесс структурной разработки состоит из трех этапов: анализ, проектирование, реализация, сопровождение. Изначально все этапы выполнялись последовательно, пока не был закончен предыдущий этап, не начинался следующий.

Однако степень сложности проектов стремительным образом росла, что и стало причиной создания нового метода разработки — объектно-ориентированного подхода.

Объектно-ориентированная парадигма — естественный эволюционный шаг, основанный на структурной парадигме и ее развитии. Объектно-ориентированная парадигма фокусируется на поведенческих и структурных характеристиках сущностей, как полных модулей. Парадигма основана на понятии класса, экземпляра класса посылки сообщения, инкапсуляции, наследования и полиморфизма.

Класс — это абстракция множества предметов реального мира, удовлетворяющая следующим двум правилам (рис. 1.1):

1. Все предметы в этом множестве — экземпляры класса (синоним экземпляра класса — объект) имеют одни и те же свойства.

2. Все объекты обладают одним и тем же набором правил и линий поведения.

Атрибут — это абстракция одного индивидуального свойства (характеристики) объекта. Атрибут имеет имя и значение. Для атрибутов должны выполняться правила: объект имеет одно-единственное значение для каждого атрибута в данный момент времени, и атрибут не должен содержать никакой внутренней структуры.

Табличная интерпретация (рис. 1.2) класса и объектов позволяет в упрощенной форме понять соотношение между этими понятиями. Од-

нако табличная интерпретация не показывает методов класса и механизмов инкапсуляции. Рис. 1.3 иллюстрирует эти механизмы.

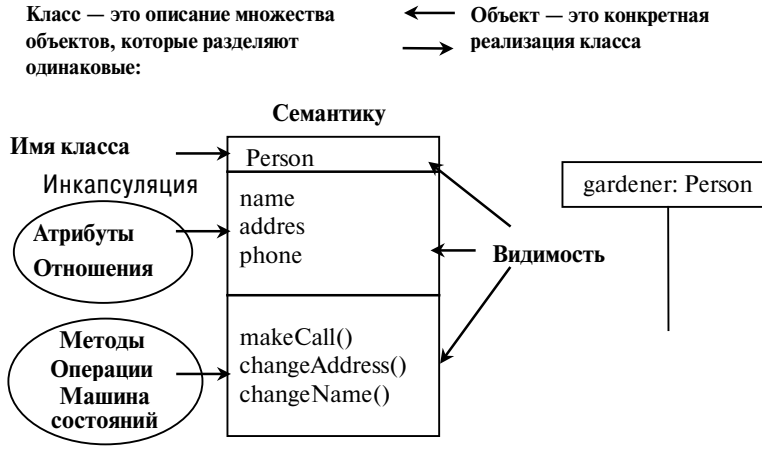


Рис. 1.1. Класс и экземпляр класса

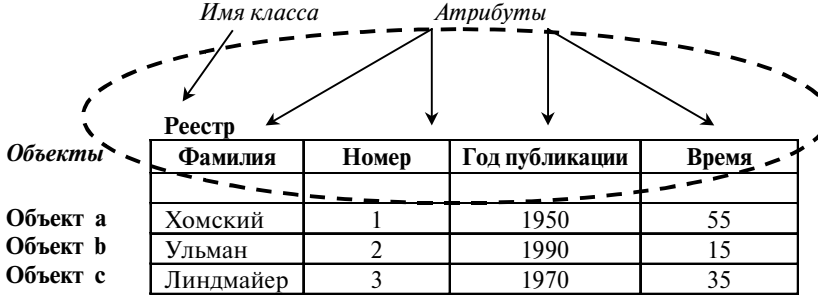


Рис. 1.2. Табличная интерпретация класса и объектов

Пример программы, соответствующий областям видимости (рис. 1.3):

```
class A {
  public:
    A(){}  // конструктор объекта без параметров и
    ~A(){} // деструктор объекта без параметров
  private:
  protected:
};
```

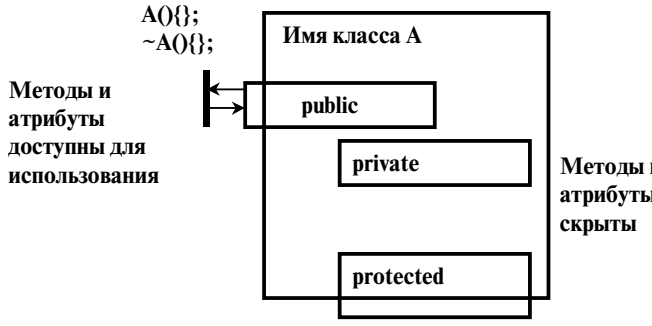


Рис. 1.3. Инкапсуляция и области видимости

Имя конструктора и деструктора объекта соответствует имени класса A. Конструкторы используются для создания объектов и инициализации переменных (присвоения им начальных значений). Например, для рассматриваемого примера программа для класса и определения объектов выглядит следующим образом. Предварительно надо подключить библиотеки для вывода на печать. Это делается с помощью резервированных слов #include, далее указывается имя библиотеки.

```
#include <iostream>
#include "string.h"
class A {
  public:
    A(){} // конструктор без параметров
    // инициализация начальных значений
    A(char * vf,int vnum, int vyear, int vold){
      f = vf;
      num = vnum;
      year = vyear;
      old = vold;
    }
    void print (){
      cout << "num = " << num << endl;
      cout << "f = " << f << endl;
      cout << "year = " << year << endl;
      cout << "old = " << old << endl;
    }
  private:
    char *f;
    int num;
    int year;
    int old;
  protected:
};
```

```
void main() {
  A a("Хомский",1,1950,55);
  a.print();
  A b("Ульман",2,1990,15);
  b.print();
  A c("Линдмайер",3,1970,35);
  c.print();
}
```

В объектно-ориентированном программировании действие инициируется при помощи передачи сообщения объекту, который принимает ответственность за выполнение определенных действий. Например, для вывода на печать в классе A описана функция print(), которая, используя объект cout библиотеки <iostream>, с помощью операторов << выводит на печать строки и значения переменных. При этом сам оператор << перегружен для разных значений параметров.

В отличие от вызова процедуры посылка сообщения имеет назначенного приемника. Например, a.print();. Интерпретация сообщения зависит от приемника и может варьироваться среди различных приемников. Например,

b.print();
c.print().

Ответственность увеличивает уровень абстракции и дает возможность еще большей независимости между объектами. Приемник берет ответственность за ответ на сообщение. Пославшему сообщение нет необходимости знать детали использования метода. Это стандартная мощная форма скрытия информации.

Объектно-ориентированная парадигма поддерживает следующие принципы: инкапсуляцию, наследование, полиморфизм. Рассмотрим эти механизмы.

Инкапсуляция держит данные и операции над этими данными вместе, изолируя проектную информацию от реализации.

Полиморфизм включает способность новых объектов быть определенными как вариации существующих объектов, вводится новая реализация, но спецификации остаются такими же, то есть для различных реализаций может существовать одна и та же спецификация (рис. 1.4).

Спецификация используется для описания, что есть объект и что объект делает.

Реализация описывает, как объект реализован.

Перегрузка имен функций (функций с идентичными именами, но разными типами параметров) и параметрический полиморфизм были реализованы в структурных языках программирования. Например, перегрузка функций в C++:

```
int add (int a, int b) {return a + b;}
float add (float x, float y) {return x + y;}
```

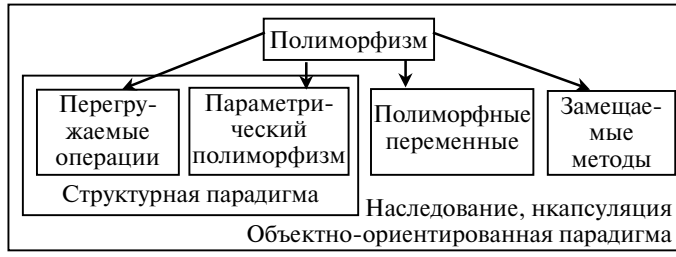


Рис. 1.4. Полиморфизм

Параметрический полиморфизм, например, в C++ представлен в форме шаблонов на основе *контейнерного* класса, элементы которого задаются как типы параметров T, вместо которых подставляются конкретные объекты. Посредством шаблонов можно определить параметрические полиморфные функции. Например, программа, использующая шаблон (template) функции swap — для пересылки двух значений, выглядит следующим образом:

```
template <class T>
void swapp (T &a, T &b){
  T t = a; a = b; b = t;
}
void main() {
  int a = 5;
  int b = 77;
  cout << "a = "<< a << endl;
  cout << "b = "<< b << endl;
  // явное объявление функции-шаблона
  swapp <int>(a,b);
  cout << "swap"<<endl;
  cout << "a = "<< a << endl;
  cout << "b = "<< b << endl;
}
```

Наследование — разделение кода: взятие и использование подобия в данных и поведении, имеет отношение и повторно использует существующие объекты, чтобы определить новые объекты. Для этого классы организуются в иерархические наследуемые структуры (см. рис. 1.5). Класс потомок (или подкласс) будет наследовать атрибуты от родительского класса (суперкласса) вышестоящего по иерархии.

В объектно-ориентированные языки программирования были добавлены: полиморфные переменные, замещаемые функции. Если базовые переменные класса принимают тип переменной производного класса — они являются *полиморфными* переменными.

Например, растения (суперкласс Plant) обладают общими свойствами и линией поведения и подразделяются на цветы (подкласс Flo-

wer), деревья (подкласс Tree). Отношение наследования изображено на рис. 1.5 стрелками. Для вывода изображения применяется функция draw(). Работа с графикой описывается в классе Draw.

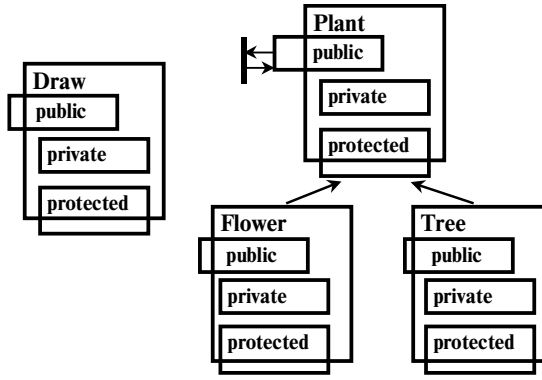


Рис. 1.5. Схема наследования классов

Программа, соответствующая схеме классов:

```
#include <iostream.h>
#include "string.h"
class Plant {
  public:
    Plant(){count = 0;}
    ~Plant(){}
    virtual void draw (){cout << count << endl;}
  private:
  protected:
    int count;
};
class Flower: public Plant {
  public:
    Flower(){count = 1;}
    ~Flower(){}
    virtual void draw (){cout << count << endl;}
  private:
  protected:
};
class Tree: public Plant {
  public:
    Tree(){count = 2;}
    ~Tree(){}
    virtual void draw (){cout << count << endl;}
  private:
  protected:
};
```

```
class Draw {
  public:
    Draw(){}
    ~Draw(){}
    void polymorph (Plant plant) {
      plant.draw();
    }
  private:
  protected:
};
void main() {
  Flower flower;
  Tree tree;
  Draw draw;
  draw.polymorph(tree);
  draw.polymorph(flower);
}
```

В классе Draw объявлена функция polymorph (Plant plant), параметром которой является переменная plant суперкласса Plant — *полиморфная* переменная, ее значениями могут быть объекты наследуемых классов. В классах Plant, Flower, Tree используется замещаемая функция draw() с ключевым словом virtual, определяющим замещение функции в зависимости от типа.

Замещаемая функция подкласса переопределяет функцию с тем же самым именем родительского класса. Функция, выполняемая в ответ на сообщение, зависит от его приемника. При замещении выполняется поиск и связывание функции. Поиск начинается с объекта получателя и продолжается вверх наследуемой цепи. Когда функция с тем же именем доступна самому верхнему классу в иерархии наследования, выполняется замещение. Замещаемая функция способна обеспечить полиморфное поведение в соответствии с типом, который переменная имеет при вызове функции, например, draw.polymorph(tree). Выполнение замещаемых функций в ответ на одно и то же сообщение является формой полиморфизма.

Объектно-ориентированные библиотеки используют рассмотренные виды полиморфизма и состоят из коллекции классов, спроектированных для разнообразных целей. Специализированные библиотеки предоставляют разработчику интерфейс программирования (API — Application Programming Interface).

Для сокращения времени разработки, кроме объектно-ориентированных библиотек, используют прототип разрабатываемой системы. Интеративная и инкрементная разработка на основе прототипа позволяет повторно применить готовые решения.

1.2. Расширение объектно-ориентированной парадигмы

Парадигма образца (pattern) — это концептуализация проблемы, ее решение, реализация или понимание решения. Цель образца — сохранить опыт проектирования ПО, избежать перепроектирования или по крайней мере минимизировать его, сделать объектно-ориентированное проектирование повторно используемым.

Образец имеет четыре существенных элемента, которые лежат в его основе для описания архитектурных проблем и решений:

- *образец имя* — это имя мы можем использовать, чтобы описать проблемы проектирования, их решения в одно или два слова;
- *контекст* — это ситуация, дающая возникновение проблемы;
- *проблема* — это периодичность возникновения ситуации, требующей решения в этом контексте;
- *решение* — это доказанное разрешение проблемы.

Компонентная парадигма введена для сокращения времени на разработку системы за счет повторного использования программно-незави-симых компонент. Понятие компонента было сформулировано на Европейской конференции по объектно-ориентированному программированию (ECOOP, где EC — аббревиатура от Europe Conference) в 1996 г.:

- «Компонента программного обеспечения — это модуль композиции с контрактно специфицируемыми интерфейсами и только с явными контекстными зависимостями. Компонента программного обеспечения может быть размещена независимо и является объектом для композиции третьих компаний».
- Инкапсулируемая часть программного обеспечения *систем* реализует специфицирующий сервис или множество сервисов. Компонента имеет один или больше *интерфейсов*, которые обеспечивают доступ к их сервисам. Компоненты служат строительными блоками для структуры системы. На уровне языка программирования компоненты могут быть представлены как *модули*, классы, объекты или множество связанных родством *функций*.

Парадигма контура (framework) — это интегрированное множество компонент, которое обеспечивает модель взаимодействия (для программного обеспечения часто выполняется приложение кода, непосредственно формируемое внутри контура). Контур предоставляет разработчику характерную архитектуру для решения проблемы.

Прототип, реализованный в контуре, сокращает время инкрементной разработки.

Существуют следующие технологии контура: OpenDoc, CORBA, .NET, Java NetBeans, Web-pages. Рассмотрим их архитектуру.

OpenDoc *архитектура* — стандарт сложного документа, предложенный в 1993 г. и принятый OMG как совместимый стандарт с CORBA. Совместимость между OpenDoc и COM+ достигается с помощью библиотек и интерфейсов. Компоненты в OpenDoc документе называются «части», а «редактор частей» манипулирует этими данными. Система Bento определяет формат контейнера, который может быть использован в файлах, сетевых потоках.

CORBA *архитектура* — (Common Request Broker Architecture — CORBA) стандарт Общей Архитектуры Брокера Объектных Запросов. Интерфейсы объектов могут быть определены и помещены в репозитарий интерфейсов двумя способами статически, т. е. описываются на языке определения интерфейсов IDL (Interface Definition Language) или динамически. В структуре ORB выделяется ядро, обеспечивающее внутреннее представление объектов и передачу заявок. Среда ORB поддерживает отображение IDL в соответствующий язык.

.NET *архитектура* — объектно-ориентированная основа платформы Windows. Сервис сложного документа управляет компонентами. Промежуточный язык позволяет разрабатывать многоплатформенные приложения.

Архитектура NetBeans — *платформа, включает* Java виртуальную машину и Java компоненты, которые унифицированы для всех операционных систем и имеют с ними соответствующие интерфейсы.

Архитектура Web *браузера* представляет собой Web Контейнер, загружающий HTML, XML страницы, которые реализуют концепцию сложного документа.

Агентно-ориентированная парадигма характеризуется этапами разработки, ориентированными на агентов, являющихся объединением объектно-ориентированной парадигмы и искусственного интеллекта. Агент управляет собственным процессом, связывается с другими агентами, с внешним миром и обновляет свою структуру. Агент может копировать самоподобных агентов для выполнения поставленной задачи, при этом выполняя роль прототипа.

Сервис-ориентированная парадигма характеризуется этапами разработки, ориентированными на сервисы (услуги), предоставляемые системой. Механизмы абстракции — сервис.

Аспектно-ориентированная парадигма подразделяет системные домены и прикладные домены. Влияет на этап проектирования и реализации.

Фрактальная парадигма — характеризуется самоподобием повторно используемых компонент, самоорганизацией и самомодифицируемостью.

Рис. 1.6 иллюстрирует эволюцию парадигм.

Рис. 1.6. Механизмы абстракции, гранулярности и жизненного цикла парадигм программирования

ГЛАВА 2

ПРИМЕНЕНИЕ КАТЕГОРИЙ ПРИ РАЗРАБОТКЕ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ОПЕРАЦИОННЫХ СИСТЕМ

В настоящей главе рассматривается концепция архитектуры ОС, приводится классификация категорий, рассматривается метод проектирования объектно-ориентированных ОС на основе категорий и контура.

2.1. Концепция архитектуры операционной системы

Архитектура ОС компьютера [1] представляет собой три взаимосвязанных описания.

1. Архитектуру (формат) множества инструкций компьютера (ISA — Instruction Set Architecture). Это описание поведения компьютера, с точки зрения системного программиста, отвечающего за ассемблер.

2. Архитектуру организации операционной системы (монолитную послойную или микроядро). Это описание операционной системы, с точки зрения архитектора ОС, отвечающего за состав компонент ОС, организацию (соединение и объединение компонент). Архитектура должна проектироваться с учетом производительности ОС.

3. Архитектуру реализации (например, структурная или объектно-ориентированная реализация). Это описание операционной системы, с точки зрения программиста, отвечающего за реализацию ОС, состав модулей и их взаимосвязь. Включает детальное логическое проектирование и специфические аспекты реализации.

Эти три описания архитектуры взаимосвязаны. Решения, сделанные в каждом из них, могут влиять на остальные. Например, бинарные фиксированные инструкции компьютера обладают меньшей гибкостью и не требуют разработки дополнительного программного обеспечения.

Архитектура реализации описывается на основе категорий, определяемых между двумя классами объектов.

2.2. Классификация категорий

Категория как математическое понятие имеет дело с абстрактными математическими структурами и отношениями между ними. Отношения между классами объектов моделируются математикой и логикой.

Такая модель коллекции объектов с некоторым структурным подобием определяется категорией [2] (см. Приложение С).

Дорожная карта — аналог категории. Карта дает больше информации о дорогах, чем о городах. Для категорий важны не сами объекты, а как мы добираемся от одного объекта к другому. Категория определяет сущность класса, состоящего из связанных математических объектов. Вместо определения индивидуальных объектов, как это делается традиционно, определяется структура отображения между объектами.

Пособие носит практическую направленность, поэтому математические аспекты категорий опущены. Применение категорий позволяет абстрагироваться на этапе проектирования от синтаксиса объектно-ориентированного языка программирования [2, 3], который будет использован для реализации.

На рис. 2.1 изображен граф категории C_R. Классы изображаются прямоугольниками, помеченными модификаторами ограничения видимости: public (прямоугольник выступает — область видимости открыта), private (прямоугольник вложен — область видимости скрыта), protected (прямоугольник выступает внизу — область видимости открыта только для отношения наследования). R_n — отношение между объектами. Объекты — изображаются метками (черными прямоугольниками). Метод или функция изображается черточками и моделирует взаимодействие объектов. Изменение состояния объекта показывается индексами объектов.

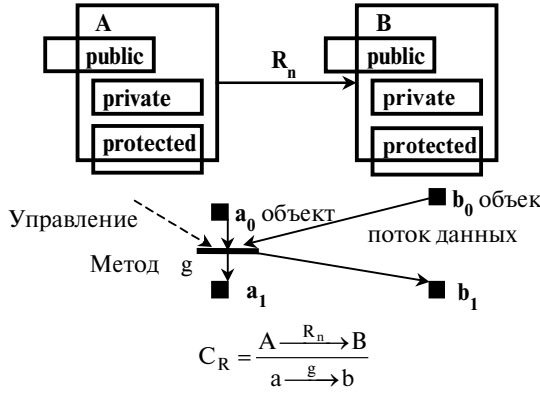


Рис. 2.1. Категория описывает динамику и статику

На входной (операция чтение) или выходной (операция запись) дуге указываются атрибуты соответственно чтения и записи. Для вы-

полнения метода необходимо, чтобы атрибуты объектов находились в начальных состояниях. Если для выполнения метода нужны все атрибуты объекта, то дуга помечается именем объекта.

Категорию, кроме графического представления, будем описывать дробным выражением. Числитель — отношение между классами (статика), знаменатель — взаимодействие и состояние объектов (динамика), g — метод. Примеры будем иллюстрировать посредством метода «сборка» (конструктора в терминах языка объектно-ориентированного программирования).

Классификация категорий основывается на пяти типах отношений $\mathbf{R_n}$ между объектами: агрегация, наследование, ассоциация, использование и конкретизация.

Категория агрегация. Отношение агрегация применяется для создания иерархии «Целое — часть», также известное как правило «has-a». Категорию агрегация будем обозначать как:

$$C_{\text{has-a}} = \frac{A \xrightarrow{\text{has-a}} B}{a \xrightarrow{g} b}.$$

Рассмотрим упрощенную структуру объекта диспетчер (целое) для управления ресурсами ОС (см. рис. 2.2). Диспетчер состоит из планировщика, осуществляющего планирование процессами и их приоритетами, блока управления памятью и загрузчика.

Отношение агрегации $R_{\text{has-a}}$ — между классами, показано графически прямоугольниками внутри класса диспетчер. Диспетчер агрегирует объекты, таким образом, что к ним имеют доступ только методы объекта диспетчер. Графически это показано пунктирной линией и словом модификатором private (скрытый).

Взаимодействие объектов посредством метода «сборка» показано диаграммой взаимодействий. При этом объект диспетчер изменяет свое состояние, а в состоянии диспетчер_2 агрегирует объекты. Пример описывается следующими категориями агрегации:

$$C^1_{\text{has-a}} = \frac{\text{Диспетчер} \xrightarrow{\text{has-a}} \text{Планировщик}}{\text{диспетчер} \xrightarrow{\text{сборка}} \text{планировщик}};$$

$$C^2_{\text{has-a}} = \frac{\text{Диспетчер} \xrightarrow{\text{has-a}} \text{Блок\_управления\_памятью}}{\text{диспетчер} \xrightarrow{\text{сборка}} \text{блок\_управления\_памятью}};$$

$$C^3_{\text{has-a}} = \frac{\text{Диспетчер} \xrightarrow{\text{has-a}} \text{Загрузчик}}{\text{диспетчер} \xrightarrow{\text{сборка}} \text{загрузчик}}.$$

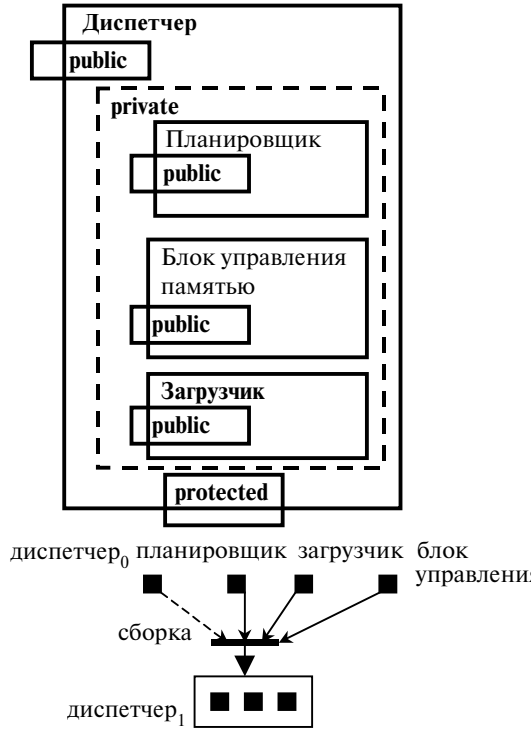


Рис. 2.2. Диспетчер управления ресурсами

Категория наследование. Отношение наследование применяется в иерархии «Общее — Частное», также известное как правило «is-a». Объекты организуются в иерархическую структуру с наследованием свойств и методов (функций и операций). Категорию наследование будем обозначать как

$$C_{\text{is-a}} = \frac{A \xrightarrow{\text{is-a}} B}{a \xrightarrow{g} b}.$$

Рассмотрим пример, иллюстрирующий категорию наследование (см. рис. 2.3). Пусть Контроллеры имеют общие алгоритмы с сетевыми и дисковыми устройствами ввода-вывода. Чтобы повторно использовать общие алгоритмы, организуем иерархию «Общее — частное», для этого воспользуемся категорией наследование.

Взаимодействие объектов посредством метода «сборка» показано диаграммой взаимодействий: объекты диск и сетевое устройство (част-

ное), являются (is-a) производными от объекта контроллер (общее). Методы взаимодействия между такой иерархией объектов должно удовлетворять принципу подстановки [2]. Пример описывается категорией наследование:

$$C_{\text{is-a}} = \frac{\text{Контроллер} \xrightarrow{\text{is-a}} \text{Сетевое\_устройство}}{\text{контроллер} \xrightarrow{\text{сборка}} \text{сетевое\_устройство}}.$$

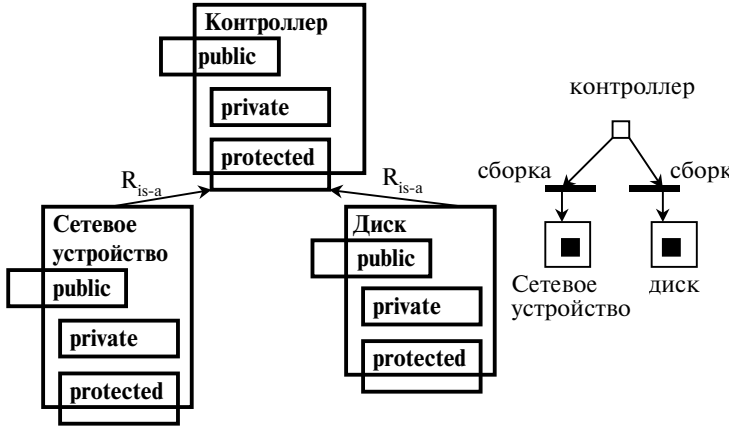


Рис. 2.3. Категория наследование

Категория ассоциация. Отношение ассоциация — смысловое семантическое отношение, указывающее семантическую двухстороннюю зависимость объектов классов A:B. Категорию ассоциация будем обозначать как

$$C_{A:B} = \frac{A \xleftarrow{C} B}{a \xleftarrow{g} b}, \text{ где C — ассоциативный класс.}$$

Рассмотрим пример. Задание—Процессор (рис. 2.4). Задания запрашивают процессоры, конкурируя с целью овладеть процессором, чтобы сформировать объект связи один к одному (1:1). Пусть конкурирующие запросы преобразуются в последовательную форму с помощью монитора, который определяет процессор для выполнения задания. Монитор — это ассоциативный объект, отвечающий за создание связи. Состояние Монитора сохраняется в модели состояний и не является атрибутом (см. рис. 2.4).

В результате выполнения метода «сборка» динамически устанавливается отношение 1:1. Пример описывается категорией ассоциация:

$$C_{1:1} = \frac{\text{Задание} \xleftarrow{\text{Монитор}} \text{Процессор}}{\text{задание} \xleftarrow{\text{монитор}} \text{процессор}}.$$

Целесообразно использовать в курсовом проекте ассоциацию с динамически изменяемыми связями, чтобы уменьшить связанность между объектами.

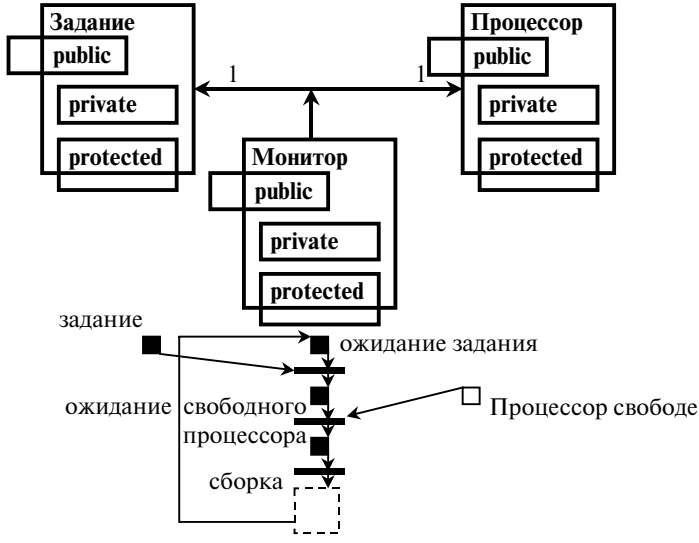


Рис. 2.4. Категория ассоциация

Категория использование. Отношение использование — одностороннее отношение между объектами: ресурс одного объекта, выступающего в качестве сервиса, используется другим объектом, выступающим в качестве клиента. Категорию использование будем обозначать со стрелкой направленной к сервису, как

$$C_{use} = \frac{A \xleftarrow{use} B}{a \xleftarrow{g} b}.$$

Например, Планировщик выполняет метод планирования процессов, готовых к выполнению (рис. 2.5).

Пример описывается категорией использование:

$$C_{use} = \frac{\text{Процесс} \xleftarrow{use} \text{Планировщик}}{\text{процесс} \xleftarrow{\text{планировать}} \text{планировщик}}.$$

Категория конкретизация. Отношение конкретизации возникает между объектами, когда класс, выступающий в качестве шаблона (temp-

late в C++, generic в Java, Ada) с параметрами, конкретизируется другим классом. В параметр(ы) шаблона подставляется класс. Категорию конкретизация будем обозначать со стрелкой, направленной к шаблону, как

$$C_{generic} = \frac{A \xleftarrow{generic} B}{a \xleftarrow{g} b}.$$

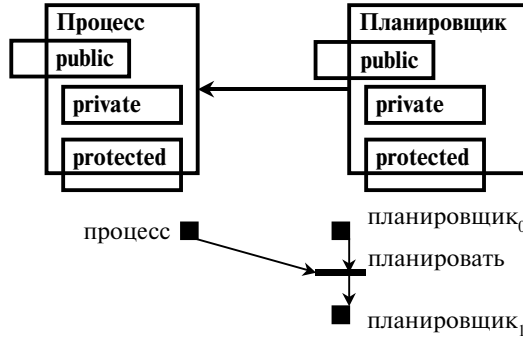


Рис. 2.5. Категория использование

Например, требуется сконструировать очередь процессов (рис. 2.6).

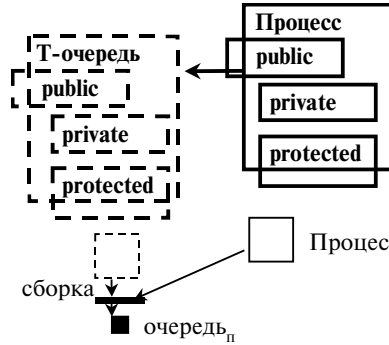


Рис. 2.6. Категория конкретизация

Пример описывается категорией конкретизация, метод сборка создает объект очередь$_п$:

$$C_{generic} = \frac{\text{Т-очередь} \xleftarrow{generic} \text{Процесс}}{\text{очередь}_п \xleftarrow{\text{сборка}} \text{Процесс}}.$$

Классификация категорий показана на рис 2.7. Каждая категория имеет варианты, оказывающие влияние на программную архитектуру ОС.

Рис. 2.7. Классификация категорий

Проект ОС описывается в терминах категорий, при этом между категориями возможны различные комбинации. Программная реализация категорий приведена в приложении С.

2.3. Проектирование на основе категорий и контура

Проектирование на основе категорий может быть применено для решения различных задач. Для того чтобы упростить понимание работы ОС и реализацию проекта собственной модели ОС, применяется контур.

Контур — это интегрированное множество компонент, которое реализует модель взаимодействия. Контур предоставляет разработчику характерную архитектуру для решения проблемы. Код контура является выполняемым.

Пусть требуется спроектировать загрузчик процессов в соответствии со схемой аппаратных и программных компонент (рис. 2.8). Процедура проектирования с использованием категорий и контура шаг 2 и 9 состоит из следующих шагов.

Шаг 1. Определить и описать классы в соответствии с заданной схемой. Выявить атрибуты классов. Идентифицировать классы и атри-

буты значащим именем. Выделим следующие классы: Память (оперативная Memory), Загрузчик (Loader), Процесс (Process) и Файл (File). Для классов определим следующие атрибуты: Процесс — идентификатор id, Файл — имя, Память — размер (рис. 2.9).

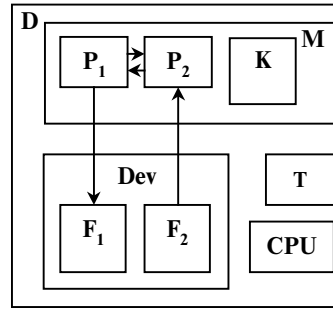


Рис. 2.8. Схема 1

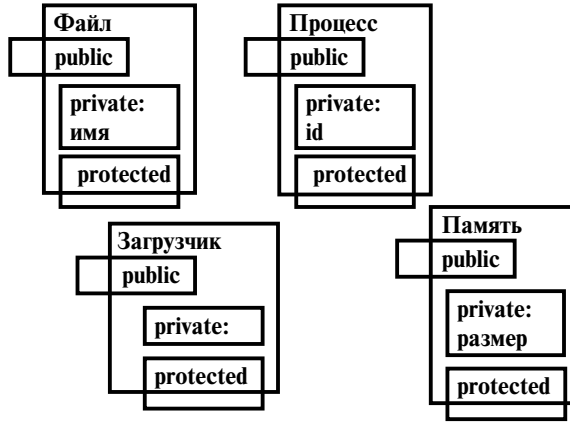


Рис. 2.9. Определение классов

Шаг 2. Изучить контур для проектирования сетевых ОС (лабораторная работа 1). Составить семантическую сеть, использующую классы контура. Например, Память. Составить семантическую сеть, выделить классы контура прямоугольником в верхнем правом углу и показать возможные отношения (рис. 2.10).

Шаг 3. Провести архитектурный анализ полученной сети (лабораторная работа 2), с точки зрения производительности. Например, может быть принято решение о введении объектов для кэш-памяти или объектов посредников для уменьшения связанности объектов.

Шаг 4. Рассмотреть по парам вершины семантической сети и определить отношения для категорий. Загрузчик считывает из указанного Файла программу и загружает ее как Процесс в Память. Рассмотрим Загрузчик как клиент, использующий сервис Файл и сервис Память, Загрузчик-Процесс — это отношение агрегация, Загрузчик — «Целое», а Процесс как «Часть». Семантическая сеть (рис. 2.10), имеет следующее описание:

$N = <$ {Файл, Загрузчик, Процесс, Память}, {R_1, R_2, R_3, R_4} >, где

$R_1 = \mathbf{R_{use}}$: Файл ← Загрузчик

$R_2 = \mathbf{R_{has\text{-}a}}$: Загрузчик → Процесс

$R_3 = \mathbf{R_{use}}$: Память → Загрузчик

$R4 = \mathbf{R_{use}}$: Память ← Процесс

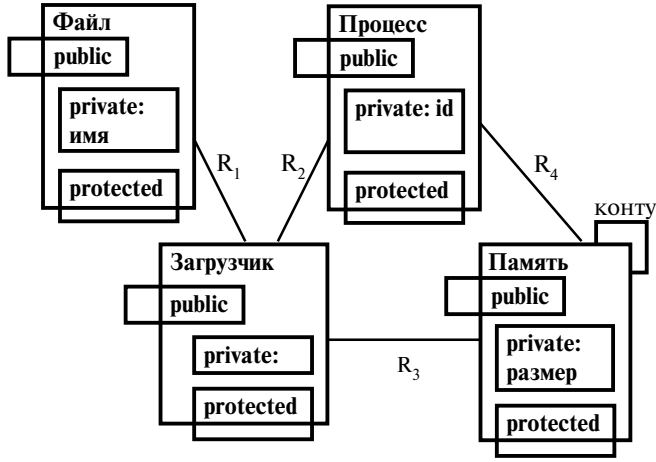


Рис. 2.10. Составление семантической сети

Шаг 5. Определить объекты и идентифицировать их. Объектам следует присваивать первые строчные буквы идентификаторов классов. Например, файл (Файл), загрузчик (Загрузчик), память (Память) и процесс (Процесс). Состояниям объектов приписываются индексы. На рис. 2.11 объектам в начальном состоянии приписаны индексы 0. Не допускайте сокращения имен объектов — это ведет к непониманию текста программы, хотя схемы вариантов выданных заданий для компактности имеют сокращения.

Шаг 6. Определить взаимодействия между объектами.

Для рассматриваемого примера Диспетчер порождает событие, которое является внешним по отношению к Загрузчику, на рис. 2.10.

показано пунктирной линией. Метод загрузить инкапсулируется в объекте Загрузчик, что показывается расположением метода, изображаемого черточкой, под классом, инкапсулирующим его.

Таким образом, во взаимодействии объектов метода загрузить участвуют объекты файл, загрузчик, память и процесс. Алгоритм метода рассматривается как *черный ящик*, то есть на этом шаге нас не интересует его реализация.

Шаг 7. Определить состояния объектов (начальные, текущие и конечные, если они есть) и условия выполнения метода.

Входящие в метод объекты находятся в состояниях, которые являются условиями выполнения метода загрузить. Например, должен быть указан файл, из которого осуществляется считывание программы, и он должен быть доступен для чтения, должна быть в наличии свободная память для размещения объекта процесс, например, на дуге, входящей в метод, указывается размер доступной памяти.

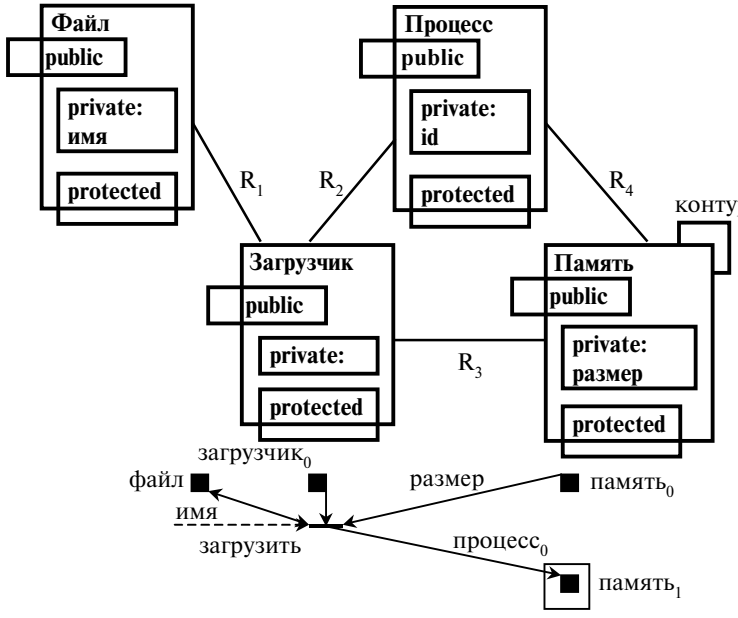


Рис. 2.11. Формирование категорий

Метод загрузить создает новый объект процесс и изменяет состояние объекта память, что показывается индексами. Выходящие из метода объекты будут находиться в текущем состоянии, например, память$_1$.

Шаг 8. Представить описание проекта в виде категорий. Следует описать сценарий методов. Например, для метода загрузить требуется реализовать четыре категории, последовательность действий алгоритма загрузить указывается верхним индексом категории:

$$C^1_{use} = \frac{\text{Файл} \xleftarrow{use} \text{Загрузчик}}{\text{файл} \xleftarrow{\text{загрузить}} \text{загрузчик}};$$

$$C^2_{has\text{-}a} = \frac{\text{Загрузчик} \xrightarrow{has\text{-}a} \text{Процесс}}{\text{загрузчик} \xrightarrow{\text{загрузить}} \text{процесс}};$$

$$C^3_{use} = \frac{\text{Память} \xleftarrow{use} \text{Загрузчик}}{\text{память} \xleftarrow{\text{загрузить}} \text{загрузчик}};$$

$$C^4_{use} = \frac{\text{Память} \xleftarrow{use} \text{Процесс}}{\text{память} \xleftarrow{\text{загрузить}} \text{процесс}}.$$

Шаг 9. Реализовать решение на объектно-ориентированном языке программирования, используя реализацию контура и категорий [2].

Взаимодействия между объектами имеют специфику для каждой категории. Входные и выходные объекты метода реализуются как входные параметры метода или доступные объекты, в зависимости от категории. Например, для категории использование — это входные параметры, для категории агрегация — объекты, доступные методу.

ГЛАВА 3

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ

Практические занятия представляют собой лабораторные работы, сгруппированные по четырем наиболее важным темам. Всего в пособии восемь лабораторных работ, рассчитанных на один семестр.

Первая тема «Архитектуры вычислительных систем» посвящена изучению архитектуры ЭВМ и архитектур операционных систем: монолитной, послойной и микроядра, проектированию для заданной схемы категорий и архитектуры мини-ОС.

Во второй теме «Управление процессами» требуется изучить модели состояний процессов и методы синхронизации процессов, применяемые в многозадачных ОС: блокирующие переменные, критические секции, мьютексы и семафоры. Многозадачная ОС должна чередовать выполнение нескольких процессов, чтобы повысить степень использования процессора, а также распределять ресурсы между процессами в соответствии с заданной стратегией.

В третьей теме «Управление оперативной и виртуальной памятью» требуется изучить метод свопинга (перемещение информации) между виртуальной и оперативной памятью, спроектировать и реализовать один из методов организации памяти: с фиксированным или динамическим распределением, постраничный, сегментный или сегментно-страничный. Для загрузки программ из файлов в основную память необходимо разработать категорию загрузчик программ. При проектировании и реализации поставленных задач должны использоваться категории контура, облегчающие процесс разработки.

Четвертая тема «Планирование» посвящена методам планирования в системах с одним процессором и многопроцессорному планированию.

Для систем с одним процессором в контуре реализованы следующие методы планирования: первым поступил — первым обслужен (FCFS — first-come-first-served), круговое или карусельное планирование (RR — round robin), выбор самого короткого процесса (SPN — shortest process next), наименьшее остающееся время (SRT — shortest remaining time), наивысшего отношения отклика (HRRN — highest response ration next), планирование с динамическим приоритетом (DP — Dynamic Priority).

Для многопроцессорного планирования в контуре реализован метод назначение процессоров (архитектура ведущий/ведомый) с распределением загрузки. В курсовых проектах в качестве заданий для разработки и реализации могут задаваться методы планирования: назначение процессоров (архитектура равноправных процессоров), разделение загрузки, бригадное планирование и динамическое планирование.

При выполнении курсового проекта операционная система рассматривается как диспетчер ресурсов, который объединяет категории на основе контура, разработанные в лабораторных работах. Целью курсового проекта является разработка и реализация сетевой ОС, поэтому основное внимание уделяется межпроцессорному взаимодействию и реализации архитектуры сетевой файловой системы.

В рамках заданного варианта двухзвенной (клиент-сервер) или трехзвенной архитектуры сети для разработки задаются методы взаимодействия компьютеров путем передачи сообщений по сети: метод удаленного вызова процедур RPC (Remote Procedure Call), метод сокетов.

Архитектура сетевой файловой системы реализуется с помощью сетевых файловых служб, взаимодействующих между собой.

Названия категорий, реализованных в контуре, соответствуют общепринятой терминологии операционных систем UNIX и Windows, что позволяет в упрощенном виде изучать и практически применять основополагающие методы ОС.

Тема № 1. Архитектуры вычислительных систем

Лабораторная работа № 1

Лабораторная работа состоит из двух частей: в первой части требуется изучить архитектуру компьютера, а во второй части, используя контур и заданную схему программных и аппаратных компонент, разработать программы на языке ассемблер, которые интерпретируются контуром.

I. В первой части требуется изучить: архитектуру компьютера, прикладной интерфейс программирования контура, алгоритм выполнения инструкций процессором и механизм прерываний, реализованный в контуре, формат инструкций для выполнения процессором контура, диаграмму и категории контура, которые применяются для реализации архитектуры компьютера.

Общая структура компонент компьютера. На рис. 3.1.1 приведена общая структура компонент компьютера, состоящая из следующих основных элементов: процессора, основной памяти, контроллера ввода-вывода и системной шины.

Процессор (CPU — Central Process Unit) осуществляет контроль над действиями компьютера, а также выполняет функцию обработки данных.

Основная память (Memory) здесь хранятся данные и программы. Как правило, эта память является временной.

Контроллер ввода-вывода (Controller) служит для передачи данных между компьютером и внешним окружением, состоящим из различных периферийных устройств, в число которых входит вторичная память, коммуникационное оборудование и терминалы.

Системная шина (Bus) определенная структура и механизмы, обеспечивающие взаимодействие между процессором, основной памятью и контроллером ввода-вывода.

Назначение и использование регистров

PC (program counter) — программный счетчик. Содержит адрес ячейки памяти, откуда следует извлечь очередную команду. Если не указано иное, то после выполнения команды увеличивается на размер команды, т. е. программа выполняется линейно. Но этот порядок может изменяться при выполнении соответствующих команд процессора.

IR (instruction registor) — регистр команд. Содержит последнюю выбранную из памяти команду.

MAR (memory address registor) — регистр адреса памяти, куда заносится адрес ячейки памяти, в которой будет производится операция чтения-записи.

MBR (memory buffer registor) — регистр буфера памяти, куда заносятся данные, предназначенные для записи в память, или те, которые будут прочитаны из нее.

IO/AR (Input/output address registor) — регистр адреса ввода-вывода, куда заносится адрес устройства ввода-вывода.

IO/BR (Input/output buffer registor) — регистр буфера ввода-вывода, служит для обмена данными между устройством ввода-вывода и процессором.

PSW (Processor Status Word) — слово состояния процессора. Этот регистр содержит биты кода состояний, которые задаются командами сравнения, режимом (пользовательский или режим ядра), и другую служебную информацию. Обычно пользовательские программы могут читать весь регистр PSW целиком, но писать могут только в некоторые из его полей. Регистр PSW играет важную роль в системных вызовах и операциях ввода-вывода.

SP (Stack Pointer) — указатель стека. Он содержит адрес вершины стека в памяти. Стек содержит по одному фрейму (области данных) для

каждой процедуры, которая уже начала выполняться, но еще не закончена. В стековом фрейме процедуры хранятся ее входные параметры, а также локальные и временные переменные, не хранящиеся в регистрах. SP позволяет работать с данными в соответствии с принципом стека. Выделение данных в стеке происходит быстрее, чем просто в области данных.

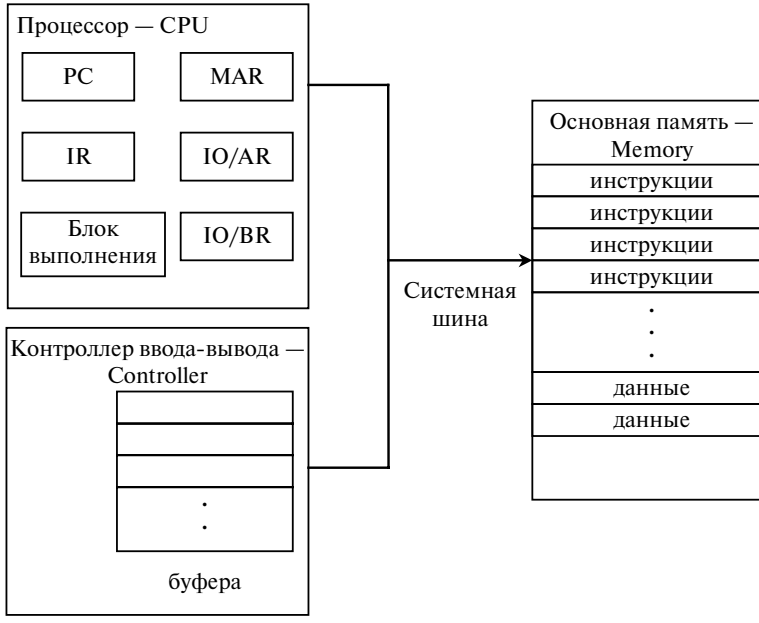


Рис. 3.1.1. Компоненты компьютера

Для поддержки модульного программирования и гибкого использования данных ОС должна выполнять такие функции, как: изоляция процессов, автоматическое размещение и управление программ в памяти, поддержка модульного программирования (динамическое создание, уничтожение и изменение их размера), защита и контроль доступа, долгосрочное хранение. ОС выполняет эти требования с помощью средств виртуальной памяти (рис. 3.1.2).

Контур для проектирования ОС описывается диаграммой категорий (рис. 3.1.3), которая включает блок управления памятью — компоненту ОС. Диаграмма составлена на основе категорий наследования, агрегации и использования.

Переменные, определяющие конфигурацию системы, объявлены в файле ArchitectureCPU.h. Рассмотрим описание и назначение классов

контура и их прикладной интерфейс программирования — API (Application Programm Interface).

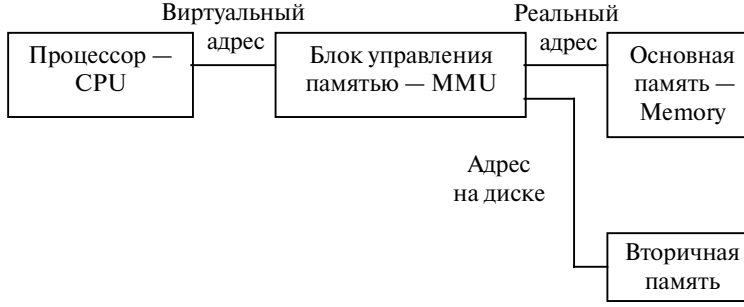


Рис. 3.1.2. Адресация виртуальной памяти

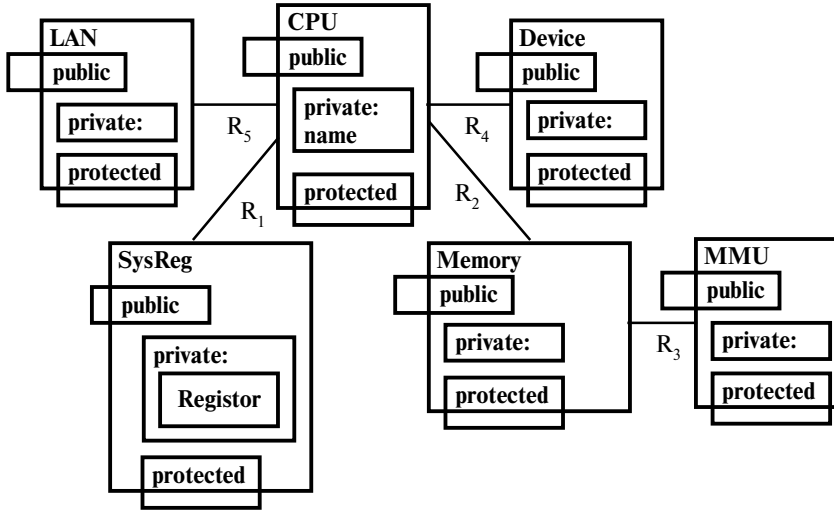


Рис. 3.1.3. Семантическая сеть, описывающая компоненты компьютера и блок управления памятью MMU ОС

Таблица 3.1.1

Описание и назначение классов

Класс /идентификатор	Назначение класса
CPU	Класс, моделирующий процессор
Memory	Класс, моделирующий оперативную память, состоит из блоков. Каждый блок — это объект класса Block. Определены функции чтения и записи по заданному адресу блоков и записи инструкции в память

Продолжение табл. 3.1.1

MMU (Memory Manager Unit — блок управления памятью)	Класс для реализации блока управления памятью разработчиками ОС. Класс содержит нереализованные функции, возвращающие реальный адрес по заданному виртуальному, и наоборот
SysReg	Класс, моделирующий регистры системы. Включает класс регистр Register. Используется для сохранения состояния регистров системы при переключении между процессами
Block	Модель инструкции CPU, по размерам равна блоку памяти

Таблица 3.1.2

Процессор **(CPU)**

CPU(Memory & memory, int MAX_PROCESS)	Конструктор класса, в качестве параметра передается модель оперативной памяти и максимальное количество процессов из main.h
Interrupt exeInstr(int addr, int id)	Выполнение одной инструкции программы, загруженной в память. Если прерывания не произошло, возвращает OK
virtual Interrupt decode(Block* block, SysReg *sysreg)	Декодирование инструкции. Функция объявлена в области protected
Block* fetch(SysReg *sysreg)	Выборка инструкции из памяти по адресу в PC. Функция объявлена в области private

Таблица 3.1.3

Класс оперативная память **(Memory)**

Memory (int SIZE_OF_MEMORY_IN_BLOCKS)	В конструкторе задается размер оперативной памяти в блоках. Размер объявляется в main.h
Block* read(int addr)	Читает из памяти по адресу addr блок памяти
void setAddrBlock(int addr)	Устанавливает начальный адрес для последовательной записи
void setCmd(int cmd, int op1, int op2)	Применяется при записи инструкций в память
Block* read(int addr)	Читает из памяти по адресу addr блок
int getAddrFreeBlock()	Выбирается первый свободный блок при просмотре и возвращается адрес блока

Таблица 3.1.4

Класс, реализующий управление памятью **(MMU)**

MMU(Memory &memory)	В конструкторе задается ссылка на память
void swapIn(ProcessImage * processImage)	Загрузка программы из образа процесса в основную память

Void swapOut(ProcessImage * processImage)	Выгрузка программы из основной памяти, применяется в алгоритмах вытеснения
virtual int getRealAddr()	Определяет свободный адрес для загрузки программы. Ограничения, проверяется только один блок, остальные считаются свободными — что не всегда так
void setAlloc(int addr)	Установить адрес размещения программы в памяти
void Debug(HANDLE * handle)	Показать процесс выполнения swapIn
void DebugMemory()	Показать состояние памяти

Таблица 3.1.5

Класс системных регистров **(SysReg)** модуль ArchitectureCPU.h

int getState(Name name)	Получить состояние регистра по имени Name, где enum Name {r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14, PC,SP}
setState(_int8 nregister,int operand)	Установить состояние регистра nregister по значению operand

Register	Класс Register имеет три атрибута:
	int numreg; // Номер регистра;
	Name name; // Имя атрибута;
	int state; // Состояние регистра

Register* register_ [NUMBER_OF_ REGISTERS];	Методы SysReg обращаются к массиву

Алгоритм выполнения инструкций процессором контура

Рассмотрим пример выполнения инструкций в процессоре (см. рис. 3.1.4).

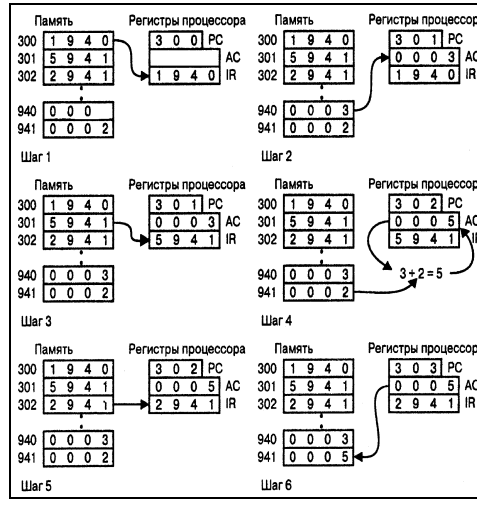
Содержимое памяти и регистров в примере представлено шестнадцатеричными числами.

Шаг 1. Адрес первой команды, хранящейся в счетчике, — 300. Эта команда (она представлена шестнадцатеричным числом 1940) загружается в регистр команд (IR), а показание программного счетчика увеличивается на 1. В этом процессе участвуют регистры адреса и буфера памяти, однако для упрощения они игнорируются.

Шаг 2. Первые 4 бит (первая шестнадцатеричная цифра) регистра команд указывает на то, что нужно загрузить значение в аккумулятор. Остальные 12 бит (три шестнадцатеричные цифры) указывают адрес 940.

Шаг 3. Из ячейки 301 извлекается следующая команда (5941), после чего значение программного счетчика увеличивается на 1.

К содержимому аккумулятора прибавляется содержимое ячейки 941, и результат снова заносится в аккумулятор.



3.1.4. Пример исполнения программы

Шаг 4. Из ячейки 302 извлекается следующая команда (5941), затем значение программного счетчика увеличивается на 1.

Рассмотрим выполнение инструкций процессором контура (рис. 3.1.5). Алгоритм выполнения инструкции имеет конвейерную архитектуру. Инструкция для выполнения вычислений может использовать 16 регистров. Для процессора контура приняты следующие ограничения:

- реализован регистр PC, регистр SP не реализован (задание на реализацию может быть выдано в качестве индивидуальной курсовой работы);
- реализованы только инструкции арифметики (задание на реализацию других инструкций может быть выдано в качестве индивидуальной курсовой работы);
- регистры и память работают с числами, это относится и к выражениям.

Алгоритм выполняется при вызове функции объекта CPU: Interrupt exeInstr(int addr, int id), где addr — адрес программы в основной памяти, id — идентификатор программы, а Interrupt — код прерывания, возвращаемый функцией.

Алгоритм иллюстрирует четыре шага выполнения программ в однозадачном режиме работы процессора.

Шаг 1. Выбор инструкции из памяти — Block* fetch (int id). Инструкция записана в блоке памяти — Block.

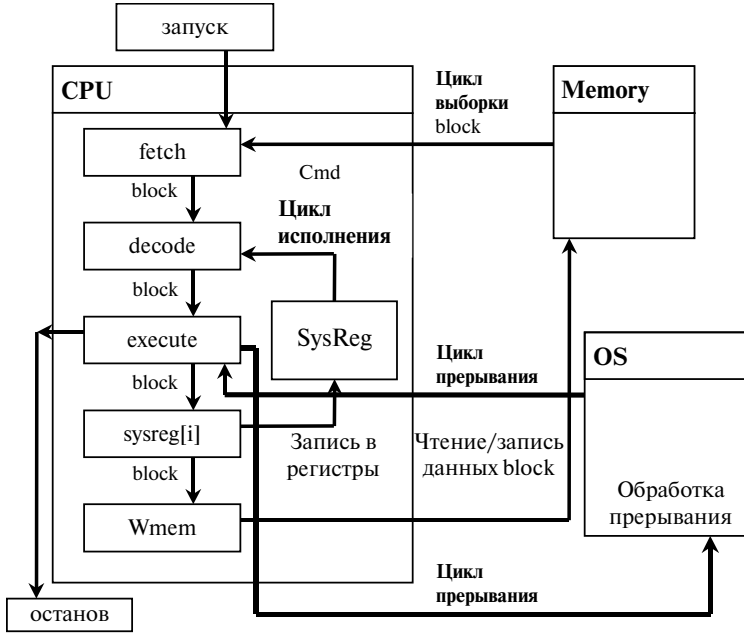


Рис. 3.1.5. Алгоритм выполнения инструкций в модели

Шаг 2. Декодирование инструкции — decode(block, id).

Шаг 3. При выполнении, инструкции осуществляется доступ к системным регистрам SysReg, реализуется как sysreg[id] -> getState (имя регистра).

Шаг 4. Выполнение инструкции: выполнение действий.

Результат выполнения может быть записан в основную память: для этого применяется инструкция: Wmem r[x],[int].

В алгоритме процессора контура (модуль CPU.h), выполняющем инструкцию, реализовано сохранение состояния регистров. Для каждого id — идентификатора процесса (программы) выделяется уникальный набор регистров, который требуется при реализации многозадачного режима, когда при переключении между процессами требуется сохранять состояние регистров:

```
// выполнить инструкцию - блок
Interrupt exeInstr(int addr, SysReg *sysreg) {
  Interrupt interrupt;
  Block* block = fetch(sysreg);
```

```
  // пустой блок не декодируется
  if (block->getState()) return Error;
  interrupt = decode(block, sysreg);   // прерывание
  switch (interrupt){
    case OK:  // изменить PC
      sysreg->setState(PC,
        memory-> getNextAddr(sysreg->getState(PC)));
      break;
    case Exit: cout << "Exit" <<endl;
      return Exit;
    case Dev: // обращение к устройству печати
      dev->printData(sysreg->
        getState((Name)block->getNregister()) );
      // изменить PC
      sysreg->setState(PC,
        memory->getNextAddr(sysreg->getState(PC)));
      break;
    case Lan:  // обращение к устройству сети
      lan->sendData(sysreg->
        getState((Name)block->getNregister()) );
      // изменить PC
      sysreg->setState(PC,
        memory->getNextAddr(sysreg->getState(PC)));
      break;
    default : return interrupt;
  }
  return OK;
} // end exeInstr
```

Механизм прерываний ОС позволяет реагировать на внешние события, происходящие асинхронно вычислительному процессу, передавая управление в устройства ввода-вывода и обработчикам ОС.

В зависимости от источника прерывания делятся на три класса:

1) внешние прерывания, связанные с сигналами от внешних устройств;

2) внутренние прерывания, возникающие в результате ошибок вычислений;

3) программные прерывания, представляющие собой удобный способ вызова процедур операционной системы.

Механизм прерываний поддерживается аппаратными средствами компьютера и программными средствами операционной системы.

Существуют два основных способа выполнения прерывания: векторный (vectored), когда в процессор передается номер вызываемой процедуры обработки прерывания, и опрашиваемый (polled), когда процессор вынужден последовательно опрашивать потенциальные источники запроса прерывания.

Для упорядочивания процессов обработки прерываний все источники прерываний распределяются по нескольким приоритетным уровням, а роль арбитра выполняет диспетчер прерываний ОС.

В контуре номер программного прерывания (перечислимый тип Interrupt, объявленный в модуле ArchitectureCPU.h) возвращается exeInstr, а прерываания устройств LAN и Dev обрабатываются в конструкции switch(interrupt): поток данных, в данном случае данные регистра пересылаются в устройства (см. алгоритм). В табл. 3.1.6 приведен список прерываний контура.

Таблица 3.1.6

Список прерываний

Название	Номер	Описание
OK	0	Нормальное выполнение программы
Error	–1	Ошибка выполнения программы
Exit	16	Программа завершает выполнение
Sys	11	Системное прерывание, обрабатываемое операционной системой
Div_0	3	Ошибка деления на ноль означает, что программа выполнила недопустимую операцию
Lan	32	Прерывание для устройства сети
Dev	256	Прерывание для устройства ввода-вывода

Список прерываний может быть расширен разработчиком. Программные прерывания должны обрабатываться ОС в отличие от прерываний устройств, например Lan и Dev.

Формат инструкции. Формат инструкций представлен внешним и внутренним описанием (рис. 3.1.6).

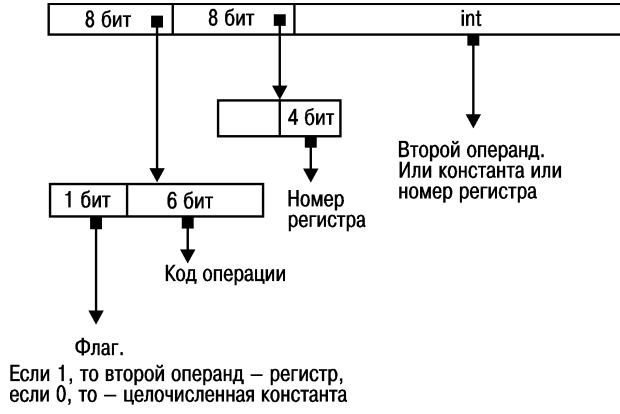


Рис. 3.1.6

Внешнее описание инструкции (язык ассемблера). Общий формат инструкции:

[команда] r[номер регистра] [$,r][номер регистра или константа].

Результат выполнения команды помещается в первый операнд.

Внутреннее представление инструкций. Программа для выполнения, размещенная в памяти, представляет собой непрерывную последовательность команд в двоичном представлении. Процессор считывает команды из памяти и выполняет. Каждая команда в памяти представляет собой две (A, B) 8-битные части команды и int-операнд (C).

В контуре размер инструкции соответствует размеру блока (Block) памяти.

A. Первая 8-битная часть A содержит код команды, задаваемый целым числом перечислимого типа Instruction. Принято соглашение, что первый операнд — регистр, второй — номер регистра или константа, что определяется спецификацией операции.

B. Вторая 8-битная часть — содержит номер регистра.

C. Последняя часть инструкции содержит в зависимости от спецификации либо константу, либо номер регистра и интерпретируется как второй операнд.

Таблица 3.1.7

Набор инструкций

Формат	Код	Описание
Арифметические, логические и побитовые операции		
Mov r[x],[int]	1	Перемещение содержимого второго операнда в первый
Add r[x],[int]	2	Сложение
Sub r[x],[int]	3	Вычитание
Mul r[x],[r]	4	Умножение
Div r[x],[r,int]	5	Деление
And r[x],[r,int]	6	Логическое И
Or r[x],[r,int]	7	Логическое ИЛИ
Xor r[x],[r,int]	8	Логическое исключающее ИЛИ
Shlu r[x],[r,int]	9	Логический сдвиг влево
Shru r[x],[r,int]	10	Логический сдвиг вправо
Shls r[x],[r,int]	11	Арифметический сдвиг влево
Shrs r[x],[r,int]	12	Арифметический сдвиг вправо
Проверка условия		
Cmp r[x],[r,int]	13	Операция сравнения
Переход		
Je [r,int]	14	Если последнее сравнение ‘=‘
Jne [r,int]	15	‘!=’
Jge [r,int]	16	‘>=’

Jgt [r,int]	17	'>'
Jle [r,int]	18	'<='
Jlt [r,int]	19	'<'
Прерывания (взаимодействие с ОС)		
Int [r,int]	20	Вызывает прерывание с указанным номером
Работа со стеком		
Pushw [r,int]	21	Положить в стек машинное слово из указанного регистра
Pushc [r,int]	22	Положить в стек байт из указанного регистра
Popw r[x]	23	Извлечь из стека машинное слово в указанный регистр
Popc r[x]	24	Извлечь из стека байт в указанный регистр
Работа с памятью. Первый операнд — что передавать в память/из памяти, второй хранит адрес, по которому сохранять/читать		
Rmem r[x],[int]	25	Загрузить из памяти слово в указанный регистр (слово блок)
Wmem r[x],[int]	26	Сохранить содержимое регистра в память как машинное слово

Процессор контура поддерживает следующие инструкции:

Mov r[x],[int], Add r[x],[int], Sub r[x],[int], Mul r[x],[r], Div r[x],[r], Int [r,int], Rmem r[x],[int], Wmem r[x],[int]

Нереализованные инструкции могут быть выданы в качестве задания для курсового проекта.

Пример программы в инструкциях, выполняемых процессором контура, демонстрируется в главной (main) программе контура (лабораторная работа 1):

```
/* Programm 1: вычисление арифметического выражения
   (10*2 + 3) / 2 – 3, вывести содержимое на печать,
   деление и умножение в модели CPU определены только между
   регистрами,
   программа в начале работы размещается по нулевому адресу */
// Перемещение содержимого второго операнда в первый (регистр)
  memory.setCmd(Mov, r1, 2);
  memory.setCmd(Mov, r2, 10);
  memory.setCmd(Mul, r1, r2); // регистр * операнд
  memory.setCmd(Add, r1, 3);
  memory.setCmd(Mov, r2, 2);
  memory.setCmd(Div, r1, r2);
  memory.setCmd(Sub, r1, 3);
  // запись содержимого r1 в память по адресу 69
  memory.setCmd(Wmem,r1, 69);
  // обращение к устройству для печати содержимого r1
  memory.setCmd(Int, r1, Dev);
  // посыл содержимого r1 по сети
  memory.setCmd(Int, r1, Lan);
  memory.setCmd(Int, 0, Exit);
```

Программы инструкций, выполняемые процессором контура, должны заканчиваться прерыванием Exit.

Диаграмма модулей и их состав. Диаграмма модулей контура приведена на рис. 3.1.7. Состав модулей приведен в табл. 3.1.8.

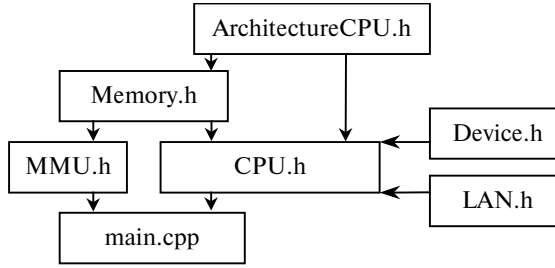


Рис. 3.1.7. Диаграмма модулей

Таблица 3.1.8

Состав модулей

Имя файла /модуля	Состав модуля
ArchitectureCPU.h	Файл содержит константы для конфигурации системы: 1. Размер оперативной памяти в блоках; 2. Максимальное количество процессов; 3. Перечисление инструкций; 4. Перечисление прерываний; 5. Количество регистров в системе; 6. Перечисление регистров. Определение класса Block, который задает формат инструкции, соответствующий размеру блока памяти. Определение класса системных регистров SysReg, который агрегирует вложением класс Register задает. SysReg задает массив регистров в соответствии с количеством регистров в системе
Memory.h	Файл содержит определение класса Memory — модель памяти компьютера в виде массива из блоков Block
CPU.h	Файл содержит определение класса CPU — модель процессора
MMU.h	Файл содержит определение класса MMU, который агрегирует класс Memory
Device.h	Файл содержит определение класса Device для вывода информации на дисплей
LAN.h	Файл содержит определение класса LAN для моделирования посылки информации по сети
main.cpp	Файл содержит главную программу main, демонстрирующую работу контура в соответствии с темами восьми лабораторных работ

II. Во второй части лабораторной работы требуется разработать две программы инструкций (для каждого процесса схемы), содержащие код прерывания при обращении к устройствам. Программы должны выполняться процессором контура (см. Приложение В. Контур для разработки сетевых ОС).

Требования к проектированию программ инструкций.

- Программы должны содержать код прерывания для обращения к устройствам в соответствии со схемой.
- Между программами должна быть организована передача информации, в соответствии со стрелками на схеме через основную память. Для этого должны быть использованы инструкции чтения из памяти и записи в память.
- Программы должны размещаться в оперативной памяти контура.
- Программы должны выполняться в контуре, а результат вычислений передаваться как данные на устройства контура.

Методические указания.

1. Программы инструкций должны состоять из простых арифметических уравнений, например 13 + 16/ 2 + 3 = 24.

2. При составлении программы используйте реализованные инструкции процессора контура.

3. Используйте пример, демонстрирующий работу контура и методы, требуемые для размещения программы в памяти и ее выполнения.

4. Организуйте дамп (dump) состояний контура с целью проверки правильности выполнения программы. Дамп — распечатка текущего состояния требуемых категорий контура.

Лабораторная работа № 2

Лабораторная работа состоит из двух частей: в теоретической части требуется изучить архитектуры ОС, а в практической части — разработать архитектуру ОС в соответствии с заданной схемой, используя одну из изученных архитектур ОС и архитектуру контура.

I. В первой части требуется изучить архитектуры ядер ОС UNIX, Solaris, Windows [5–7]. Выявить их достоинства и недостатки.

Ядро — центральная часть операционной системы, обеспечивающая приложениям координированный доступ к ресурсам компьютера, таким как процессорное время, оперативная память, внешнее оборудование. Обычно предоставляет сервисы файловой системы.

1. **Монолитная архитектура ядра** проектируется как единое целое, поэтому ОС плохо переносимо между разными платформами, так как требует внесения изменений.

2. **Многослойная архитектура ядра** ОС проектируется по слоям. Рис. 3.2.1 иллюстрирует следующие слои:

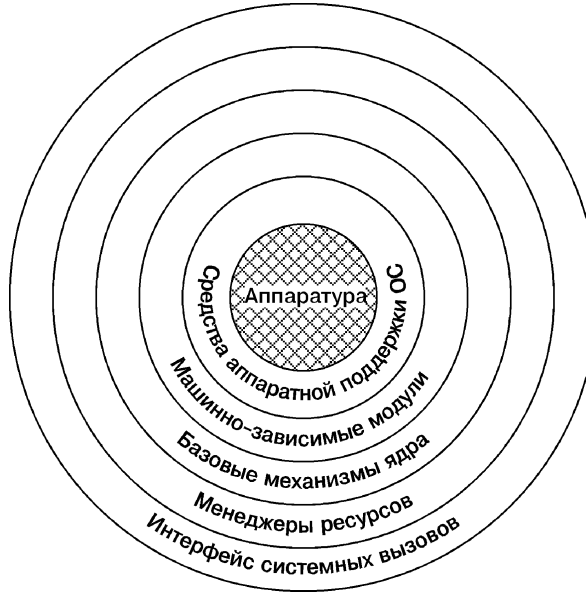


Рис. 3.2.1. Многослойная архитектура ОС

- слой машинно-зависимых компонент полностью экранирует вышестоящие слои ядра от особенностей аппаратуры. Позволяет разрабатывать вышележащие слои на основе машинно-зависимых модулей;
- слой базовых механизмов ядра выполняет операции: переключение контекстов процесса, диспетчеризация прерываний, перемещение страниц из памяти на диск и обратно;
- слой менеджеров ресурсов реализует стратегические задачи по управлению ресурсами. Каждый из менеджеров ведет учет свободных и занятых ресурсов определенного типа и планирует их распределение;
- слой интерфейс системных вызовов, реализует взаимодействие непосредственным приложением.

Достоинство многослойной архитектуры ОС — это скорость работы. Недостатки: плохая переносимость между разными платформами и расширяемость. Примеры: традиционные архитектуры UNIX (рис. 3.2.2), Solaris (рис. 3.2.3), BSD, Linux.

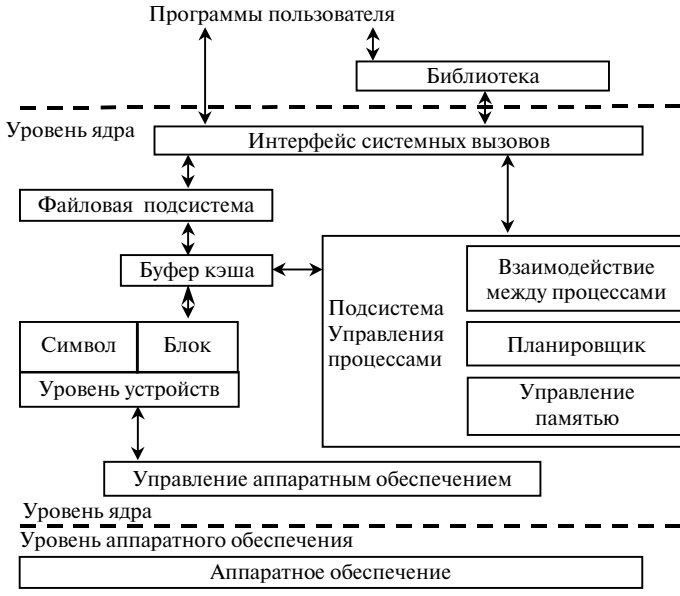


Рис. 3.2.2. Архитектура ОС UNIX

Рис. 3.2.3. Архитектура ОС Solaris

3. **Микроядерные** ОС отличаются тем, что в привилегированном режиме работает лишь малая часть ядра, называемая микроядром, все остальные функции ядра оформляются в виде приложений и работают в пользовательском режиме.

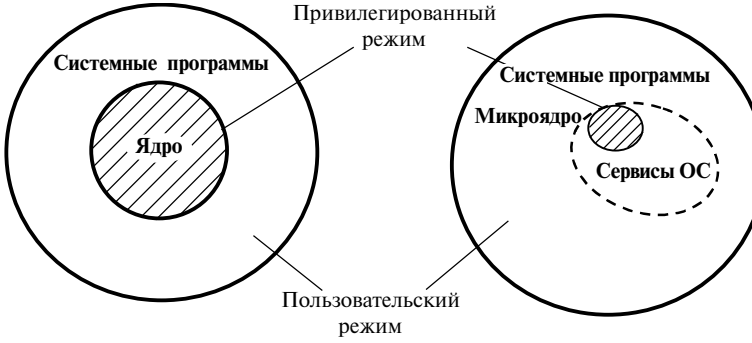


Рис. 3.2.4. Перенос основного объема функций ядра в пользовательское пространство

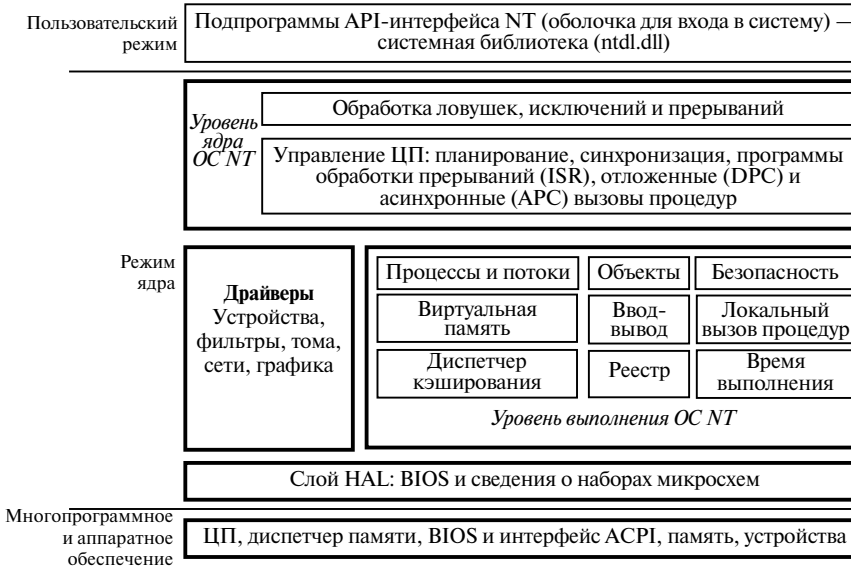


Рис. 3.2.5. Архитектура ОС Windows

Рис. 3.2.4 иллюстрирует отличия микроядерных ОС от многослойных. Пунктирной линией обведены сервисы, которые выделяются из

слоев послойной архитектуры и выполняются в пользовательском режиме для архитектуры микроядра.

Достоинства микроядерных ОС: устойчивость к сбоям оборудования, ошибкам в компонентах системы, переносимость, расширяемость. Недостатки: передача данных между процессами требует накладных расходов, снижается общая производительность при увеличении одновременно работающих сервисов. Примеры: Windows (модифицируемая архитектура микроядра [5, 7]) рис. 3.2.5, QNX, Mach, AIX, Minix, ChorusOS.

II. Во второй части лабораторной работы требуется разработать диаграмму категорий ОС в соответствии с заданной схемой и выбранным вариантом архитектуры. Варианты:

1) монолитная архитектура;
2) послойная архитектура;
3) архитектура микроядра.

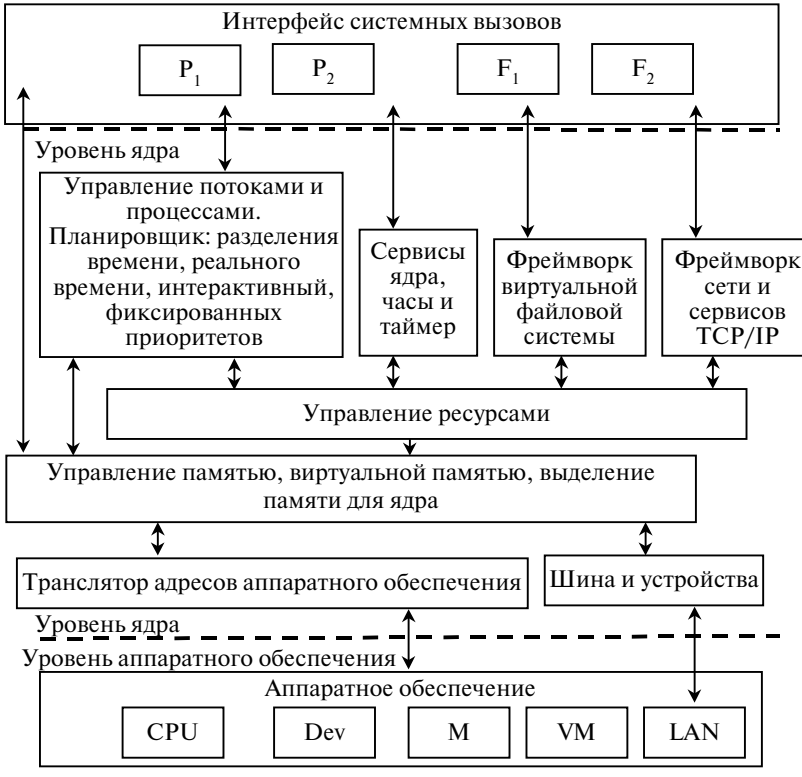


Рис. 3.2.6. Проект архитектуры ОС

При разработке необходимо учесть категории контура (см. разд. 2.2. Шаг 1–4).

Требования к проектированию архитектуры ОС.

- Архитектура должна быть спроектирована в соответствии с заданным вариантом схемы.
- Архитектура должна содержать только те категории, которые необходимы для реализации схемы.
- Один из возможных вариантов многослойной архитектуры на основе ОС Solaris приведен на рис. 3.2.6.
- Описание архитектуры должно включать диаграмму категорий, на которой должны быть помечены категории контура.

Методические указания.

- При разработке диаграммы категорий целесообразно ограничиться описанием только отношений между классами. Составленная диаграмма будет использоваться в следующих лабораторных работах, где необходимо будет определить взаимодействия между объектами и состояние объектов категории.

Тема № 2. Управление процессами

Лабораторная работа № 3

Лабораторная работа состоит из двух частей: изучение моделей процессов, очередей и разработки варианта модели процесса, включая его реализацию на основе категорий контура.

I. В первой части требуется изучить модели процессов, методы управления процессами, соответствующие категории контура.

Управление процессами является основной задачей мультипрограммной операционной системы, которая должна распределять между ними ресурсы, предоставлять процессам информацию для совместного использования и обмена, защищать ресурсы, используемые одним процессом от их использования другими процессами, и обеспечивать возможность синхронной работы процессов. Для этого операционная система должна поддерживать для каждого процесса свою структуру данных, в которой дается состояние данного процесса и указываются ресурсы, которыми он владеет. Такая структура данных называется управляющим блоком процесса (process control block — PCB).

Информация в этих блоках считывается и/или модифицируется почти каждым модулем ОС, включая модули планирования, распределения ресурсов, обработки прерываний, контроля и анализа.

В многозадачной однопроцессорной системе несколько различных процессов могут выполняться, чередуясь один с другим. В многопро-

цессорной системе несколько процессов могут не только чередоваться, но и выполняться одновременно.

Многие современные ОС поддерживают многопоточность, когда принадлежность ресурсов остается атрибутом процесса, а сам процесс представляет собой множество параллельно выполняющихся потоков. Процесс рассматривается как заявка на потребление всех видов ресурсов, за исключением процессорного времени. Процесс — это единица ресурсов. Процессорное время распределяется ОС между потоками, представляющими собой последовательности команд. Поток — единица работы.

Доступ пользователя к объектам уровня ядра, например, файлам, процессам, потокам, событиям ОС, разделам и т. д., должен быть запрещен. Контур предоставляет на уровне пользователя объект HANDLE, который является аналогом объекту, используемому в ОС Windows (см. рис. 3.3.1).

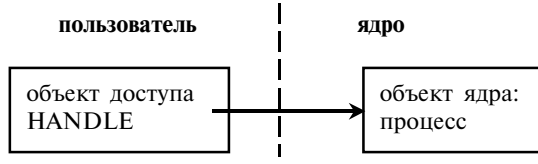


Рис. 3.3.1. Непрозрачный объект доступа HANDLE к объектам ядра

Процесс состоит из трех компонент (рис. 3.3.2):
1) выполняемой программы (кода программы инструкций);
2) ассоциированных данных, нужных программе;
3) выполняемого контекста программы, состояния регистров.

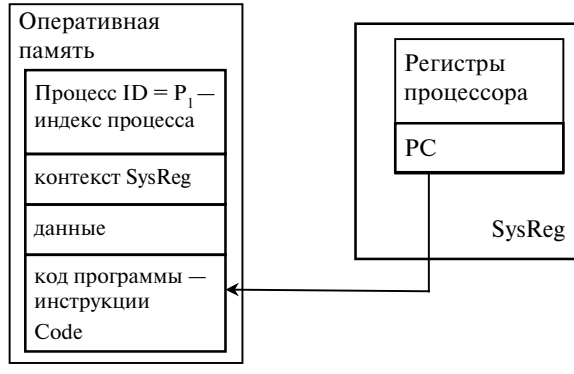


Рис. 3.3.2. Типовая реализация процесса

Процесс также называют задачей или выполнением отдельной программы. Поведение процесса можно охарактеризовать, последовательно перечислив выполняемые в ходе его работы инструкции. Такой перечень выполненных инструкций процесса называется его следом или трассировкой (trace).

Процесс обычно включает множество программ, которые должны выполняться, глобальные и локальные данные, константы и системный стек, применяемый для организации вызова процедур. *Реализация системного стека может быть выдана как задание к курсовому проекту.*

Таблица 3.3.1

Процесс (Process)

Process ()	Конструктор класса
Void setState(State state) State getState()	Изменение состояния процесса
State state	Модель состояния процесса, перечисляемый тип, заданный в модуле ArchitectureCPU.h В классе Process определен в области private

Информация, необходимая для управления процессом ОС, содержится в управляющем блоке процесса — PCB, который состоит из трех составляющих:

1) идентификации процесса, например, идентификатор ID;

2) информации о состоянии процессора — SysReg;

3) информации об управлении процессом, например размещение в памяти (addr — адрес).

1. При идентификации процесса применяются: числовые идентификаторы, которые могут быть сохранены вместе с процессом, идентификатор этого процесса, идентификатор процесса, который создал этот процесс, т. е. процесс родитель и идентификатор пользователя. *Реализация механизма организации подпроцессов может быть выдана как задание к курсовому проекту.*

2. Информация о состоянии процессора включает содержимое регистров процессора (SysReg) и слово состояний процессора (Program status word — PSW), которое используется ОС при возобновлении прерванного процесса. *Реализация механизма PSW может быть выдана как задание к курсовому проекту.*

3. Информация об управлении процессом нужна ОС для управления расписанием в зависимости от реализуемых методов. Она включает:

- состояние процесса State, например NotRunning — невыполняющийся, Running — выполняющийся, Blocked — блокированный. См. класс **Process**;

- приоритет процесса priority, используется в методе планирования процесса;
- идентификацию события, ожидаемого процессом, перед возобновлением его работы;
- структуру данных для процессов в очереди — queue. Все процессы для соответствующего состояния организуются в контуре в очередь;
- межпроцессорное взаимодействие: различные флаги — flags, сигналы — signals и сообщения — messages;
- привилегии процесса (Privileges), т. е. доступ: к памяти, утилитам, сервисам, а также выполнению типов инструкций;
- управление памятью включает ссылки на сегменты или страницы, которые описывают виртуальную память (virtual memory), выделенную процессу;
- могут определяться ресурсы, контролируемые и используемые процессом, например открытый файл. История использования ресурсов может быть тоже включена для составления расписания.

Таблица 3.3.2

Управляющий блок процесса (PCB)

PCB ()	Конструктор класса. При инициализации процессу устанавливается состояние Not Running
void setSysReg(SysReg *sysReg) SysReg * getSysReg()	Процессу присваивается выполняемый контекст программы — SysReg и размещение процесса в памяти
int getAddr() void setAddr(int addr)	Адрес размещения в оперативной памяти
int getVirtualAddr() void setVirtualAddr(int virtualAddr)	Адрес размещения в виртуальной памяти
int getTimeSlice() void setTimeSlice(int timeSlice)	Задает процессу непрерывный период процессорного время — квант для выполнения, используется при планировании в алгоритме квантования время
int getPriority() void setPriority(int priority)	Задает процессу приоритет, используется в методе планирования по приоритетам

Образ процесса (класс ProcessImage) — это категория наследования, в которую входят управляющий блок процесса (PCB), Process и HANDLE, а также категория агрегации для кода программы — Code.

Процесс может выполняться в двух режимах: пользовательском (менее привилегированный режим, обычно в этом режиме выполняют-

ся программы пользователя) и режиме ядра или системном режиме (более привилегированный режим).

Выполнение операционной системы может рассматриваться как процесс пользователя, выполняющийся в привилегированном режиме. На рис. 3.3.3 стек ядра включен в образ процесса. Другой подход — выполнение ядра, как отдельной сущности, которая оперирует в привилегированном режиме.

Рис. 3.3.3. Образ процесса: операционная система выполняется как процесс пользователя в привилегированном режиме

Таблица 3.3.3

Образ процесса (ProcessImage)

ProcessImage ()	Конструктор класса
Memory* getCode() setCode(Memory* memory)	Установка в адресное пространство, программы — инструкций (Code). Для программ инструкций выделяется память для хранения в блоках. Категория Code — категория наследования
void setStatus(bool status) bool getStatus()	Статус образа процесса, используется только при выделении VirtualMemory, которая измеряется в образах процесса. При выделении памяти: true ProcessImage свободен
void setFlag(int flag) int getFlag()	Флаг для моделирования доступа и ожидания (блокирования) ресурса требуемому процессу для выполнения

Для программного кода (Code) ОС создает образ процесса в виртуальной памяти. Код только тогда может выполняться, когда загружен в основную память.

На рис. 3.3.4 приведена диаграмма образа процесса, использующая категорию наследования.

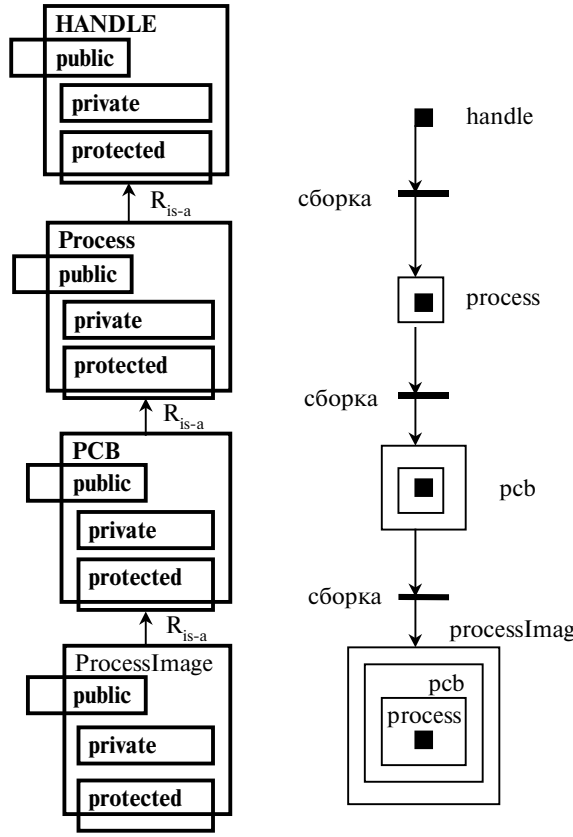


Рис. 3.3.4. Категория образ процесса ProcessImage

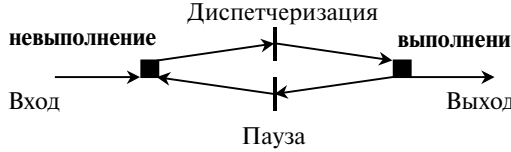
Метод создать процесс HANDLE * CreateProcess(string user, Memory *code), категории Kernel контура выполняет следующие функции:

- присваивает уникальный идентификатор ID процессу;
- выделяет и освобождает виртуальную память для образа процесса;
- инициализирует управляющий блок процесса (PCB);
- устанавливает необходимые соединения, например добавляет новый процесс в очередь NotRunning процессов;
- возвращает объект HANDLE для программирования процесса.

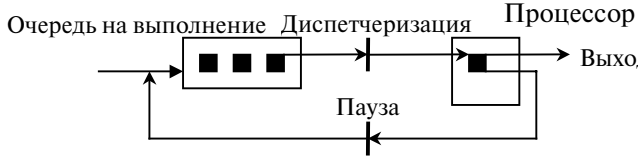
Методы, изменяющие состояние процесса в контуре, относятся к категории Dispatch и реализуют следующие функции:

- сохраняют и восстанавливают контекст процессора для процесса, используя категорию SysReg;
- перемещают указатель на образ процесса в соответствующую очередь, например из очереди NotRunning в очередь Running;
- выбирают процесс для выполнения, метод dispatch ();
- обновляют управляющий блок процесса, изменяя его состояние и временные характеристики.

Переключение процессов выполняется в течение максимально разрешенного отрезка времени. Для этого процессору устанавливается время прерывания между процессами — целое число соответствует одному циклу. Функция void setSliceCPU(int sliceCPU) категории Scheduler устанавливает время выполнения CPU, по истечении которого управление передается в вызывающую функцию.



А. Изменение состояний объекта процесс



В. Очередь объектов к процессору

Рис. 3.3.5. Модель процесса с двумя состояниями

Модель процесса (потока) с двумя состояниями. Простую модель процесса можно построить, исходя из того, что в любой момент времени процесс либо выполняется, либо не выполняется, то есть процесс может находиться в двух состояниях: выполнения и невыполнения (рис. 3.3.5, А).

Невыполнение (NotRunning) — создав новый процесс, ОС вводит его в систему в состояние невыполняющегося, однако процессор занят выполнением другого процесса. Созданный процесс ждет, пока не сможет быть запущен, образуется очередь (рис. 3.3.5, В).

Выполнение (Running) — активное состояние процесса, во время которого процесс обладает всеми необходимыми ресурсами и непосредственно выполняется процессором.

Время от времени выполняющиеся процессы прерываются и диспетчер (категория Dispatch) выбирает для выполнения другой процесс. Выполняющийся перед этим процесс переходит в состояние невыполняющийся, а в состояние выполняющийся перейдет один из ожидающих процессов.

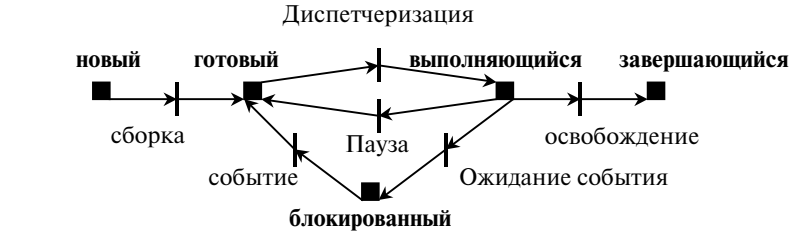
Модель с пятью состояниями. Естественно было бы разделить все невыполняющиеся процессы на два типа: готовые к выполнению и заблокированные, что облегчает диспетчеру поиск в очереди процесса для выполнения. Полезными оказываются еще два состояния: новый и завершающийся. Первое из них соответствует процессу, который был только что создан, процессу присваивается идентификатор, формируется его образ, а ОС не готова к его выполнению или имеет ограничения на количество одновременно выполняемых процессов для того, чтобы не снижать производительность системы или не переполнять память. Процесс в таком состоянии размещается в виртуальной памяти. Выход процесса из системы также происходит в два этапа: при завершении, внутренних ошибках или когда его останавливает другой процесс, имеющий соответствующие полномочия, процесс переходит в состояние завершающегося. После этого момента процесс не может больше выполняться, однако информация о процессе сохраняется и может быть использована, например, при учете процессорного времени и других ресурсов. После этого процесс удаляется из системы.

В этой модели процесс может находиться в пяти состояниях: новый, готовый, выполняющийся, блокированный, завершающийся (см. рис. 3.3.6).

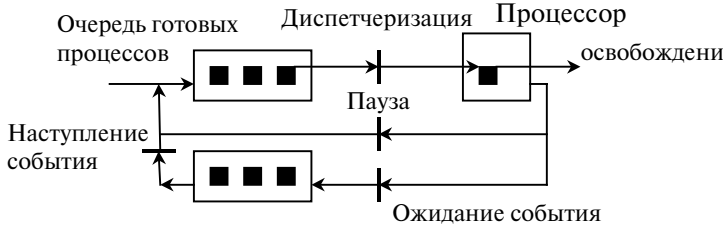
Новый (New) — для выполнения программы создается новый процесс.

Готовый (Ready) — операционная система переводит процесс из состояния новый в состояние готового к выполнению, когда процесс имеет все требуемые для него ресурсы, готов выполняться, однако процессор занят выполнением другого или других процессов.

Выполняющийся, или выполнение (Running), — когда наступает момент выбора процесса (определяется при планировании), операционная система выбирает готовый процесс на выполнение. После выполнения отведенного процессу промежутка времени или появления процесса с более высоким приоритетом ОС переводит процесс в состояние готовый.



А. Изменение состояний объекта процесс



В. Очередь объектов к процессору

Рис. 3.3.6. Модель процесса с пятью состояниями

Блокированный (Blocked) — процесс переводится в это состояние, если для продолжения работы требуется наступление некоторого события, например завершения операции ввода-вывода, получения сообщения от другого процесса или освобождения какого-либо необходимого ему ресурса. Когда возникает ожидаемое событие, процесс из этого состояния переходит в состояние готовый.

Завершающийся (ExitProcess) — если процесс сигнализирует об окончании своей работы или происходит его аварийное завершение, ОС прекращает его выполнение. Процессы из других состояний могут перейти в это состояние, если другие процессы, связанные с данным, завершились.

Состояния *выполняющийся* и *блокированный* могут быть отнесены к задачам, выполняющимся в однопрограммном режиме, а состояние *готовый* характерно только для режима мультипрограммирования.

Процессы в состоянии *готовый* и *блокированный* образуют очереди соответственно готовых и блокированных процессов.

В ходе жизненного цикла каждый процесс переходит из одного состояния в другое в соответствии с алгоритмом планирования процессов, реализуемым в данной операционной системе.

Очереди процессов представляют собой указатели на образы процессов, объединенные в списки. Такая организация очередей позволяет

легко их переупорядочивать, включать и исключать процессы, переводить процессы из одного состояния в другое.

Модель с девятью состояниями. Модель с девятью состояниями реализована в ОС Unix (см. рис. 3.3.7 и табл. 3.3.4).

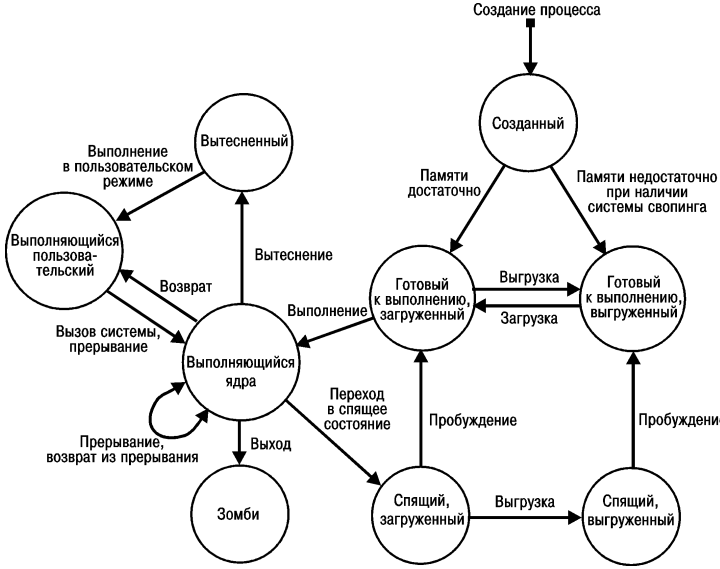


Рис. 3.3.7. Модель процесса с девятью состояниями

Таблица 3.3.4

Состояния процессов ОС UNIX

Выполняющийся пользовательский (User Running)	Выполняется в режиме пользователя
Выполняющийся ядра (Kernel Running)	Выполняется в режиме ядра
Готовый к выполнению, загруженный (Ready to Run, in Memory)	Готов выполниться в соответствии с расписанием
Спящий, загруженный (Asleep in Memory)	Не может выполняться до наступления события. Процесс в основной памяти в заблокированном состоянии
Готовый к выполнению, загруженный (Ready to Run, Swapped)	Процесс готов к выполнению, но помещается в основную память, перед тем как ему будет дана команда на загрузку в основную память в соответствии с планированием
Спящий, выгруженный (Sleeping, Swapped)	Процесс ожидает наступления события и размещен в виртуальной памяти

Продолжение табл. 3.3.4

Вытесненный (Preempted)	Процесс возвращается из режима ядра в режим пользователя, но ядро вытесняет его и выполняет переключение процесса в соответствии с планированием
Созданный (Created)	Создан новый процесс, не готовый к выполнению
Зомби (Zombie)	Процесс больше не существует, но он оставляет запись для его родительского процесса

II. Во второй части требуется разработать диаграмму для модели состояний контура и диаграмму для одной из модели состояний. Варианты:

1) модель процесса с пятью состояниями;
2) модель процесса с семью состояниями;
3) модель процесса с девятью состояниями UNIX.

Реализовать диаграмму, используя категории контура. Для заданной схемы выполнить трассировку процесса, используя демонстрационную программу контура. Определить требуемые категории и дополнить ими описание проекта архитектуры ОС.

Требования к разработки модели состояний процесса.

- Должен быть продемонстрирован свопинг программ инструкций между виртуальной и оперативной памятью, загрузка в виртуальную память и освобождение оперативной памяти, выгрузка из виртуальной памяти и ее освобождение и загрузка в оперативную память.
- Должны быть продемонстрированы состояния очередей, отражающие модель состояния процесса.
- Разработанная модель состояний должна быть включена в диаграмму категорий архитектуры ОС.

Методические указания.

- Диаграмма для модели состояний контура должна являться прототипом для разрабатываемого варианта.
- Для подготовки тестов используйте главную программу main контура.
- Для демонстрации организуйте дамп — распечатку содержимого памяти (см. main контура).

Лабораторная работа № 4

Лабораторная работа состоит из двух частей: изучения методов синхронизации процессов и практического применения методов синхронизации для процессов заданной схемы.

I. В первой части требуется изучить методы синхронизации процессов: критическая секция, блокирующая переменная, семафоры, мьютексы и мониторы.

Во время выполнения процессам часто приходиться взаимодействовать друг с другом. Примером может служить передача данных одним процессом другому или ситуация, когда несколько процессов обрабатывают один файл. Во всех этих случаях возникает проблема синхронизации процессов, которая может решаться приостановкой и активизацией процессов, организацией очередей, блокированием и освобождением ресурсов.

Рассмотрим методы синхронизации процессов: критическая секция, блокирующая переменная, семафоры, мьютексы и мониторы.

Критическая секция. Важным понятием синхронизации процессов является понятие критической секции. *Критическая секция* — это та часть программы, в которой осуществляется доступ к разделяемым данным.

Без применения особых методов синхронизации по отношению к некоторому ресурсу со стороны процессов может возникнуть эффект гонки, при котором в критической секции в данный момент времени будет находиться более чем один процесс. Прием синхронизации, решающий данную проблему, называется *взаимным исключением*. Простейший способ обеспечить взаимное исключение — позволить процессу, находящемуся в критической секции, запретить все прерывания. Однако этот способ непригоден, так как опасно доверять управление системой пользовательскому процессу; он может надолго занять процессор, а при крахе процесса в критической области крах потерпит вся система, потому что прерывания никогда не будут разрешены.

В контуре реализована синхронизация процессов с использованием методов критической секции:

вход в критическую секцию EnterCriticalSection(handle_1);
выход из критической секции kernel.LeaveCriticalSection(handle_2).

В демонстрационной версии приведен пример несинхронизированных процессов, обращающихся к общей памяти, и пример синхронизации процессов критической секцией. Фрагмент программы синхронизации:

```
// создать процесс
handle_1 = kernel.CreateProcess("P1",code_1);
// создать процесс
handle_2 = kernel.CreateProcess("P2",code_2);
kernel.EnterCriticalSection(handle_1);
kernel.EnterCriticalSection(handle_2);
// выполнение программы P1
cout << " ---------- P1 dispatch ----------" <<endl;
dispatcher.dispatch(); // изменяет состояние системы 1-й шаг
kernel.DebugProcessImage(handle_1); // состояние процессов
kernel.DebugProcessImage(handle_2); // состояние процессов
kernel.LeaveCriticalSection(handle_1);
```

```
// выполнение программы P2
cout << " --------- P2 dispatch ---------" <<endl;
dispatcher.dispatch(); // изменяет состояние системы 2-й шаг
// виртуальная память освобождена
kernel.DebugProcessImage(handle_2);
memory.debugHeap();
kernel.LeaveCriticalSection(handle_2);
cout << " dispatch" <<endl;
dispatcher.dispatch(); // изменяет состояние системы 3-й шаг

handle_1->ProcessTime();
handle_2->ProcessTime()
kernel.TerminateProcess(handle_1); // завершить процесс
kernel.TerminateProcess(handle_2); // завершить процесс
```

Блокирующая переменная обеспечивает взаимное исключение (рис. 3.4.1). С каждым разделяемым ресурсом связывается двоичная переменная, которая принимает значение 1, если ресурс свободен (то есть ни один процесс не находится в данный момент в критической

Рис. 3.4.1. Реализация критических секций с использованием блокирующих переменных

секции, связанной с данным процессом), и значение 0, если ресурс занят. Важно отметить, что операция проверки и установки блокирующей переменной должна быть неделимая (атомарная), выполняться как единое целое и не допускать разбиения на подзадачи, иначе принцип взаимного исключения может быть нарушен.

При ожидании процессом освобождения занятого ресурса происходит постоянная циклическая операция проверки блокирующей переменной, то есть процесс занимает процессорное время. Данная проблема решается при помощи *событий*. Задаются функции *Wait(x)* и *Post(x)*, где x — идентификатор некоторого события. Если ресурс занят, то процесс не выполняет циклический опрос, а вызывает системную функцию *Wait(D)*, здесь *D* обозначает событие, заключающееся в освобождении ресурса *D*. Функция *Wait(D)* переводит активный процесс в состояние *ожидание* и делает отметку в его дескрипторе о том, что процесс ожидает события *D*. Процесс, который в это время использует ресурс *D*, после выхода из критической секции выполняет системную функцию *Post(D)*, в результате чего ОС просматривает очередь ожидающих процессов и переводит процесс, ожидающий события *D*, в состояние *готовность*.

Семафоры. В 1965 г. Дейкстра (E. W. Dijkstra) ввел два обобщающих примитива для синхронизации процессов P и V. Примитивы оперируют над целыми неотрицательными переменными, называемыми *семафорами*. Пусть S такой семафор. Определим операции:

V(S) — переменная S увеличивается на 1 единым действием. Выборка, наращивание и запоминание не могут быть прерваны. К переменной S нет доступа другим потокам во время выполнения этой операции;

P(S) — уменьшение S на 1, если это возможно. Если S=0 и невозможно уменьшить S, оставаясь в области целых неотрицательных значений, то в этом случае поток, вызывающий операцию P, ждет, пока это уменьшение станет возможным.

Успешная проверка и уменьшение также являются неделимой операцией.

Рассмотрим использование семафоров на классическом примере взаимодействия двух типов потоков, выполняющихся в режиме мультипрограммирования (рис. 3.4.2).

Потоки писателя записывают данные в буферный пул, а потоки читателя считывает их из буферного пула. Данные соответствуют буферу, каждый из которых может содержать одну запись. Буферный пул состоит из N буферов. Введем два семафора e и f , где e — число пустых

буферов, f — число заполненных буферов. В начальном состоянии e = N, а f = 0.

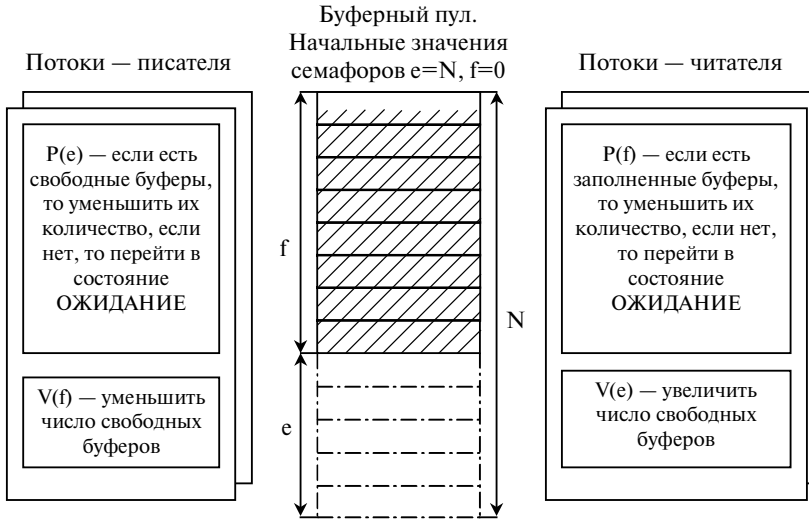


Рис. 3.4.2. Использование двух семафоров для синхронизации потоков

В общем случае потоки читателя и потоки писателя могут иметь различные скорости и обращаться к буферному пулу с переменной интенсивностью. Скорость записи может превышать скорость чтения, и наоборот. Для правильной работы потоки писателя приостанавливаются, когда все буферы оказываются занятыми, и активизируются при освобождении хотя бы одного буфера. Напротив, потоки читателя приостанавливаются, когда все буферы пусты, и активизируются при появлении хотя бы одной записи.

Если переставить местами операции P(e) и P(f), то при некотором стечении обстоятельств эти два процесса могут взаимно заблокировать друг друга. Действительно, пусть «писатель» первым войдет в критическую секцию и обнаружит отсутствие свободных буферов; он начнет ждать, когда «читатель» возьмет очередную запись из буфера, но «читатель» не сможет этого сделать, так как для этого необходимо войти в критическую секцию, вход в которую заблокирован процессом «писателем». Мы получили замкнутый круг взаимоблокировки.

Мьютекс (mutex) — это объект синхронизации, который устанавливается в особое сигнальное состояние, когда не занят каким-либо потоком. Только один поток владеет этим объектом в любой момент време-

ни, отсюда и название таких объектов, одновременный доступ к общему ресурсу исключается. После всех необходимых действий мьютекс освобождается, предоставляя другим потокам доступ к общему ресурсу.

Монитор — это набор процедур, переменных и структур данных, предложенный в 1974 г. Хоаром (Hoare) и Хансеном (Brinch Hansen) для синхронизации более высокого уровня.

Процессы могут вызывать процедуры монитора, но не имеют доступа к внутренним данным монитора. Только один процесс может быть активным по отношению к монитору. Когда процесс вызывает процедуру монитора, то первые несколько инструкций этой процедуры проверяют, не активен ли какой-либо другой процесс по отношению к этому монитору. Если да, то вызывающий процесс приостанавливается, пока другой процесс не освободит монитор. Исключение входа нескольких процессов в монитор реализуется не программистом, а компилятором, что делает ошибки менее вероятными.

II. Во второй части лабораторной работы требуется реализовать синхронизацию процессов для заданной схемы, используя категории контура. Разработать диаграмму асинхронного и синхронного системных вызовов, с помощью которых процессы приложения получают обслуживание со стороны ОС (реализуются на основе механизма программных прерываний).

В качестве заданий для курсового проекта могут быть рассмотрены методы синхронизации процессов. Варианты синхронизации процессов:

1) введением блокирующей переменной;
2) введением семафоров;
3) введением мьютексов;
4) введением мониторов.

Требования к реализации синхронизации процессов.

- Должны быть реализованы два варианта выполнения процессов, обращающихся к общему ресурсу (данным основной памяти): асинхронное выполнение и с синхронизацией процессов.
- Должны быть продемонстрированы различия выполняемых процессов в зависимости от синхронизации.
- Требуется разработать диаграммы системных вызовов, соответствующие процессу синхронизации и показать трассировку процессов.

Методические указания.

- Используйте пример лабораторной работы 4 контура.
- Диаграммы асинхронного и синхронного вызовов (рис. 3.4.3 и 3.4.4).

Асинхронный системный вызов не приводит к переводу процесса в режим ожидания после выполнения некоторых начальных системных действий, например, запуска операции ввода-вывода, управление возвращается прикладному процессу (рис. 3.4.3).

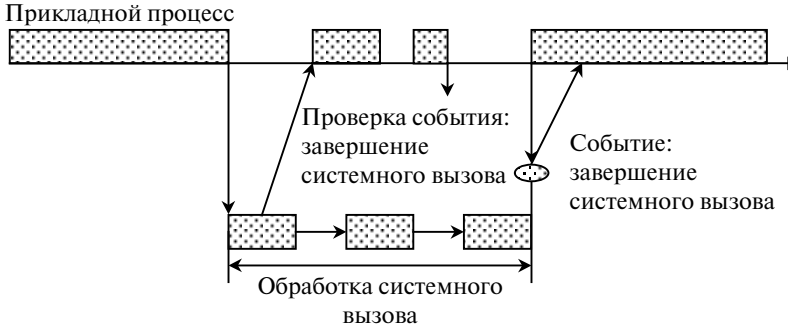


Рис. 3.4.3. Асинхронный системный вызов

Синхронный системный вызов — процесс, сделавший вызов, приостанавливается (переводится в состояние ожидания) до тех пор, пока системный вызов не выполнит всю требующуюся от него работу (рис. 3.4.4).

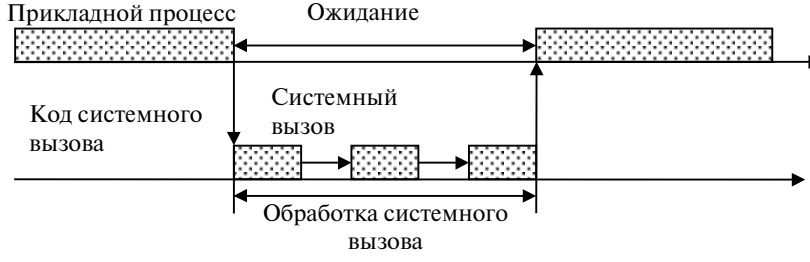


Рис. 3.4.4. Синхронный системный вызов

Тема № 3. Управление основной и виртуальной памятью

Лабораторная работа № 5

Лабораторная работа состоит из двух частей: изучения организации и методов управления оперативной памятью и практической части, в которой требуется разработать категорию загрузчик и реализовать модель виртуальной памяти.

I. В первой части требуется изучить организацию оперативной памяти: с фиксированным и динамическим распределением, со страничной и сегментной организацией.

Рис. 3.5.1 иллюстрирует отображение между типами памяти. Метод загрузить выполняется между внешней памятью, определяемой как класс Файл, объекты которого содержат программы инструкции, и оперативной памятью.

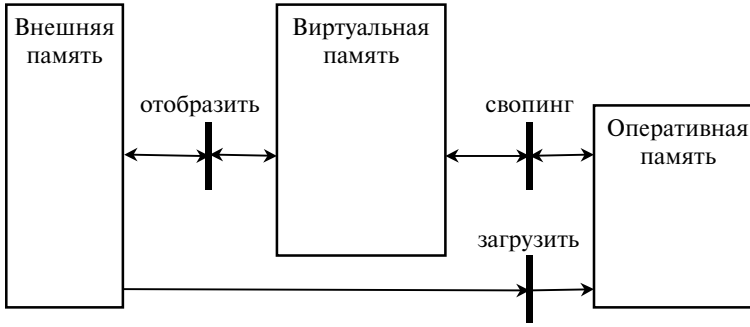


Рис. 3.5.1. Отображения между типами памяти

Загрузчик должен взаимодействовать с блоком управления памятью (MMU) контура, который должен поддерживать заданный вариант организации памяти.

Фиксированное распределение — оперативная память разделяется на ряд статических разделов во время генерации системы. Процесс может быть загружен в раздел равного или большего размера.

Блок управления памятью выполняет следующие задачи:

- сравнивая размер программы, поступившей на выполнение, и свободных разделов, выбирает подходящий раздел;
- осуществляет загрузку программы и настройку адресов.

В каждом разделе может выполняться только одна программа, поэтому уровень мультипрограммирования заранее ограничен числом разделов, число разделов не зависит от размера программы.

Недостаток — неэффективное использование памяти из-за *внутренней фрагментации* (остаются неиспользуемые участки памяти), фиксированное максимальное количество активных процессов.

Динамическое распределение — разделы создаются динамически; каждый процесс загружается в раздел строго необходимого размера. Каждому вновь поступающему процессу выделяется необходимая память. Если достаточный объем памяти отсутствует, то процесс не при-

нимается на выполнение и стоит в очереди. После завершения процесса память освобождается, и на это место может быть загружен другой процесс. Таким образом, в произвольный момент времени оперативная память представляет собой случайную последовательность занятых и свободных участков (разделов) произвольного размера.

Блок управления памятью MMU должен выполнять следующие задачи:

- ведение таблиц свободных и занятых областей, в которых указываются начальные адреса и размеры участков памяти;
- при поступлении нового процесса — анализ запроса, просмотр таблицы свободных областей и выбор раздела, размер которого достаточен для размещения поступившего процесса;
- загрузка процесса в выделенный ему раздел и корректировка таблиц свободных и занятых областей;
- после завершения процесса корректировка таблиц свободных и занятых областей.

Недостаток — *внешняя фрагментация памяти*, наличие большого числа несмежных участков свободной памяти очень маленького размера. Процедура «сжатия», выполняемая после завершения каждого процесса, — способ борьбы с фрагментацией, что ведет к неэффективному использованию процессора.

Простая страничная организация — оперативная память распределена на ряд кадров разного размера. Каждый процесс распределяется на некоторое количество страниц равного размера и той же длины, что и кадры. Процесс загружается путем загрузки всех его страниц в доступные, но не обязательно последовательные кадры. Отсутствует внешняя фрагментация.

Блок управления памятью должен выполнять следующие задачи:

- ведение таблиц свободных и занятых кадров, в которых указываются начальные адреса и размеры участков памяти;
- при поступлении нового процесса — анализ процесса и распределение его по страницам, просмотр таблицы доступных кадров и выбор кадров, размер которых достаточен для размещения поступивших страниц процесса;
- загрузка процесса в выделенные ему, не обязательно смежные, кадры и корректировка таблиц свободных и занятых кадров;
- после завершения процесса корректировка таблиц свободных и занятых кадров.

Недостаток — наличие небольшой внутренней фрагментации.

Простая сегментация. Каждый процесс распределен на ряд сегментов. Процесс загружается путем загрузки всех своих сегментов в дина-

мические (не обязательно смежные) разделы. Оперативная память распределена на ряд кадров разного размера. Отсутствует внутренняя фрагментация.

Блок управления памятью выполняет следующие задачи:

- ведение таблиц свободных и занятых разделов, в которых указываются начальные адреса и размеры участков памяти;
- при поступлении нового процесса — анализ процесса по сегментам, просмотр таблицы свободных разделов и выбор раздела, размер которого достаточен для размещения сегмента процесса;
- загрузка сегментов процесса в выделенные ему, не обязательно смежные, разделы и корректировка таблиц свободных и занятых разделов;
- после завершения процесса корректировка таблиц свободных и занятых разделов.

II. Во второй части требуется разработать категорию загрузчик, используя категории контура и заданный вариант организации основной памяти. Варианты:

1) фиксированное распределение;
2) динамическое распределение;
3) простая страничная организация;
4) простая сегментация.

Требования к реализации категории загрузчик.

- Загрузчик должен считывать программу в символьном формате, преобразовывать символы во внутреннее представление инструкции, а затем размещаться в соответствии с заданной организацией памятью, посредством блока управления памятью.
- Категория загрузчик должна включать вариант организации основной памяти и метод загрузка;
- Должны быть разработаны тесты, демонстрирующие загрузку программ инструкций в оперативную память.
- Должны быть продемонстрированы режимы в соответствии с организацией памяти: загрузка в смежные участки памяти и загрузка в распределенные участки памяти.

Методические указания.

- Разработку загрузчика программ инструкций следует проводить в два этапа: спроектировать для заданной организации памяти блок управления (OMMU, префикс O для оперативной памяти), затем спроектировать категорию загрузчик.
- За единицу измерения для всех видов организации памяти принимается блок инструкции (см. класс Block в описании контура, например, страница — 4 блока, кадр — 2–4 блока).

- При проектировании блока управления памятью используйте категорию наследование между блоком MMU контура и OMMU. Блок управления должен наследоваться от блока MMU и расширять его функции.
- При разработке загрузчика процессов в память из файла используйте пример из разд. 2.2, шаг 1–9. Сделайте определение категории и включите его в описание диаграммы категорий ОС. При разработке тестового примера используйте блок языка программирования { } в котором объявляется категория загрузки (см. лабораторную работу 8 демонстрационной программы контура).
- Для подготовки тестов используйте демонстрационный пример контура (лабораторная работа 5).

Пример программы, считывающей по символу из файла example.txt:

```
char *fileName1 = "example.txt";
ifstream ifile(fileName1);
if (! ifile.is_open()) {
  cout << "Unable to open in file"<< endl;
  return;
}
while (!ifile.eof()) {
  char ch;
  ifile.get(ch);
  cout << " konstruktor Content of ch: "<< ch << endl;
}
ifile.close();
```

Лабораторная работа № 6

Лабораторная работа состоит из двух частей: изучения методов организации виртуальной памяти и разработки категории блока управления, включающей метод свопинга (обмен, перекачка) программ инструкций между оперативной и виртуальной памятью, на основе контура.

I. В первой части требуется изучить методы организации виртуальной памяти: страничную, сегментную и сегментно-страничную организацию.

На разных этапах жизненного цикла программы для представления переменных и кодов требуются три типа адресов: символьные (имена, используемые программистом), виртуальные (условные числа, вырабатываемые компилятором) и физические (адреса фактического размещения в оперативной памяти, рис. 3.6.1).

- *Символьные имена (метки)* присваивает пользователь при написании программы на алгоритмическом языке или ассемблере.

- *Виртуальные адреса*, называемые иногда *математическими*, или *логическими* адресами, вырабатывает транслятор, переводящий программу на машинный язык. Поскольку во время трансляции в общем случае не известно, в какое место оперативной памяти будет загружена программа, то транслятор присваивает переменным и командам виртуальные (условные) адреса, обычно считая по умолчанию, что начальным адресом программы будет нулевой адрес.
- *Физические адреса* соответствуют номерам ячеек оперативной памяти, где в действительности расположены или будут расположены переменные и команды.

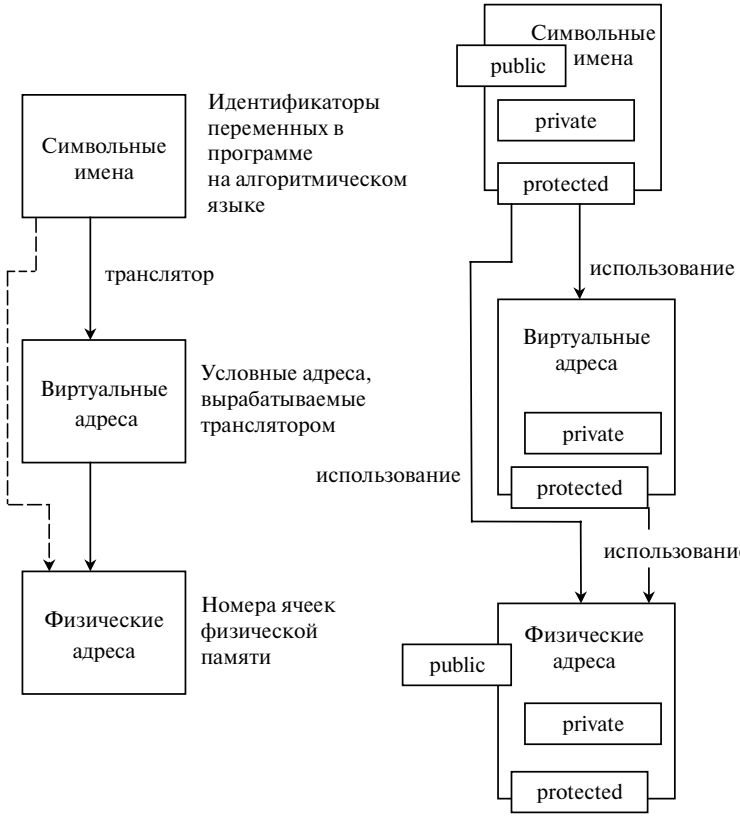


Рис. 3.6.1. Типы адресов

Совокупность виртуальных адресов процесса называется *виртуальным адресным пространством*. Диапазон возможных адресов виртуального пространства у всех процессов является одним и тем же.

Виртуальное адресное пространство может быть плоским (линейным) или структурированным.

Необходимо различать максимально возможное виртуальное адресное пространство процесса, которое определяется только разрядностью виртуального адреса и архитектурой компьютера, и назначенное (выделенное) процессу виртуальное адресное пространство, состоящее из набора виртуальных адресов, действительно нужных процессу для работы.

Виртуальное адресное пространство процесса делится на две непрерывные части: системную и пользовательскую. Системная часть является общей для всех процессов, в ней размещаются коды и данные операционной системы.

Виртуальная память — это память, которой пользователь оперирует как единым адресным пространством, но которая в действительности физически разобщена.

Виртуальная память обеспечивает очень эффективную многозадачность и облегчает работу пользователя, снижая жесткие ограничения относительно объема оперативной памяти, выделяемой прикладной программе ОС. Физическое расположение виртуальной памяти на реальных носителях может не совпадать с логической адресацией данных в прикладной программе. Преобразование логических адресов программы в физические адреса запоминающих устройств обеспечивается ОС и аппаратными средствами.

Виртуальный (логический) адрес представляет собой пару «номер страницы (сегмента) — смещение».

Страничная организация виртуальной памяти не требует одновременной загрузки всех страниц процесса. Нерезидентные страницы автоматически загружаются в память. Отсутствует внешняя фрагментация, большое виртуальное адресное пространство.

Сегментация виртуальной памяти не требует одновременной загрузки всех сегментов процесса. Необходимые нерезидентные сегменты автоматически загружаются в память.

Недостатки этих методов — накладные расходы из-за сложности системы управления памятью.

Блок управления виртуальной памятью для этих методов должен выполнять следующие задачи:

- вести таблицы свободных и занятых страниц (сегментов), в которых указываются начальные адреса и размеры участков памяти;
- при поступлении новой страницы (сегмента) просматривать таблицы доступных страниц (сегментов) и выбирать страницы (сегментов) для размещения поступивших страниц процесса. Преоб-

разовывать физический адрес страницы (сегмента) процесса в виртуальный (логический);

- корректировать таблицы свободных и занятых страниц (сегментов);
- после выгрузки страницы (сегмента) преобразовывать виртуальный (логический) адрес в физический адрес страницы (сегмента) процесса, корректировать таблицы свободных и занятых страниц (сегментов).

Сегментно-страничная организация виртуальной памяти требует разбиения на сегменты виртуального адресного пространства, а сегментов — на страницы, причем единицей перемещения образов процесса является страница.

На рис. 3.6.2 представлена сегментно-страничная модель виртуальной памяти UNIX.

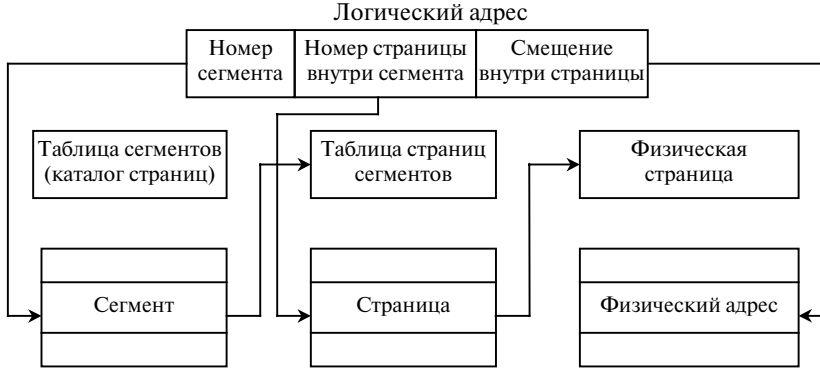


Рис. 3.6.2. Логический адрес сегментно-страничной организации

Блок управления виртуальной памятью выполняет аналогичные задачи, рассмотренные для предыдущих методов, но страничная организация и адресация поддерживается внутри сегмента, а единицей перемещения является страница.

В контуре реализована категория виртуальная память (VirtualMemory), которая реализует следующие функции (см. табл. 3.6.1).

II. Во второй части требуется реализовать вариант организации виртуальной памяти для кода программы и метод свопинга между виртуальной и оперативной памятью в соответствии с заданной организацией, используя категории контура. Варианты:

1) страничная организация;
2) сегментная организация;
3) сегментно-страничная организация.

Разработанная категория должна дополнить диаграмму категорий ОС.

Таблица 3.6.1

Виртуальная память (VirtualMemory)

VirtualMemory(int SIZE_OF_VIRTUAL_MEMORY_IN_IMAGES)	Конструктор класса. В качестве параметра задается размер виртуальной памяти в образах процесса. Каждый образ процесса (ProcessImage) имеет указатель на код программы, который измеряется в блоках памяти. Эта память должна быть структурирована в соответствии с выбранным методом организации виртуальной памяти
Void setAddrImage(int addr)	Применяется при считывании инструкций. Функция устанавливает начальный адрес для последовательной записи
ProcessImage * getImage()	Получить по установленному адресу образ процесса
void setMemory(Memory * memory)	Применяется при инициализации образа по установленному адресу, записать в виртуальную память программу
Memory* read(int addr)	Применяется при организации свопинга: из виртуальной памяти в оперативную
int getAddrFreeImage()	Выбирается первый свободный образ при просмотре

Требования к реализации категории.

- Должны быть продемонстрированы режимы работы в соответствии с вариантом организации памяти: загрузка в виртуальную память и освобождение оперативной памяти, выгрузка из виртуальной памяти и ее освобождение и загрузка в оперативную память.
- Должны быть разработаны тесты, демонстрирующие свопинг программ инструкций между виртуальной и оперативной памятью в соответствии с заданной организацией виртуальной и основной памяти.
- Для подготовки тестов используйте главную программу main контура и организуйте дамп — распечатку содержимого памяти (см. лабораторную работу 6 контура).

Методические указания.

- За единицу изменения для всех вариантов организации виртуальной памяти (как и для основной) принимается блок инструкции (см. класс Block в описании контура), например страница — 2 блока, сегмент — 4 блока.

- Разработать категорию для заданной организации виртуальной памяти VirtualMemoryCode, которая наследуется от класса Memory контура. Выполнить соответствующее замещение методов, если это необходимо.
- При создании процесса программы должны загружаться в объект VirtualMemoryCode и там храниться до окончания выполнения процесса.
- Разработать метод свопинг, который использует вариант организации памяти.
- При проектировании метода свопинга используйте пример из разд. 1.2, шаг 1–9 и категорию MMU контур.
- Разработать категорию виртуальный адрес VirtualAddress (см. рис. 3.6.1), как пара «номер страницы (сегмента) — смещение».
- Определение категорий включите в диаграмму категорий ОС.
- Для подготовки тестов используйте демонстрационный пример контура (лабораторная работа 6).

Тема № 4. Планирование

Лабораторная работа № 7

Лабораторная работа состоит из двух частей: изучения методов планирования системы с одним процессором и проведения сравнительного анализа применения методов планирования к процессам заданной схемы, используя контур.

I. В первой части требуется изучить методы планирования процессора и категории контура, поддерживающие планирование системы с одним процессором:

1. Первым поступил — первым обслужен (FCFS — first-come-first-served);
2. Круговое или карусельное планирование (RR — round robin);
3. Выбор самого короткого процесса (SPN — shortest processs next);
4. Наименьшее остающееся время (SRT — shortest remaining time);
5. Наивысшего отношения отклика (HRRN — highest response ration next);
6. С динамическим (со снижением) приоритетом (DP — Dynamic Priority).

В многозадачных системах периоды использования процессора каждым процессом чередуются с ожиданием процесса операций ввода-вывода или некоторых внешних событий. Процессор занят выполнением одного процесса, в то время как остальные находятся в состоянии ожидания.

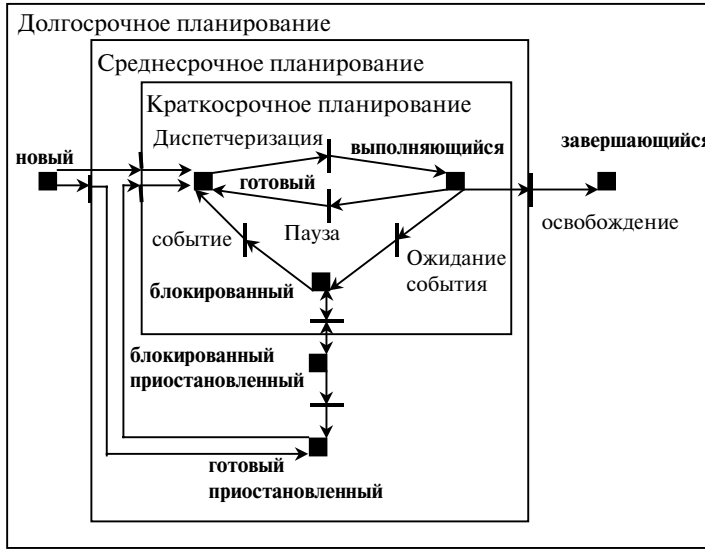


Рис. 3.7.1. Уровни планирования

Для организации многозадачности с максимальной загрузкой процессора применяют планирование. Цель планирования процессора состоит в распределении во времени процессов, выполняемых процессором для удовлетворения требованиям системы, таким как время отклика, эффективность загрузки и пропускная способность процессора.

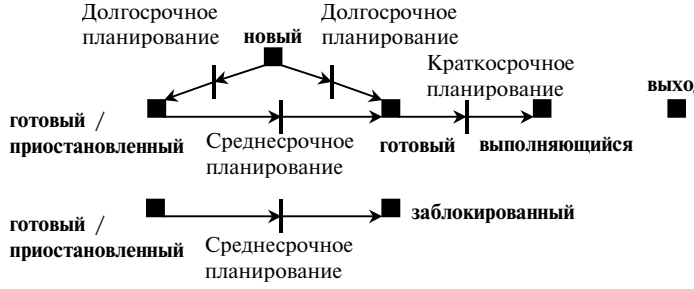
Различают три типа планирования: долгосрочное, среднесрочное, краткосрочное, а также планирование ввода-вывода. Типы планирования привязаны к диаграмме состояний объекта процесса. На рис. 3.7.1 показывается вложенность функций планирования.

Планирование оказывает большое влияние на производительность системы и решает следующие задачи: определение, какой из процессов будет выполняться, а какой ожидать выполнения; определение момента времени для смены выполняемого процесса; выбор процесса на выполнение из очереди готовых процессов.

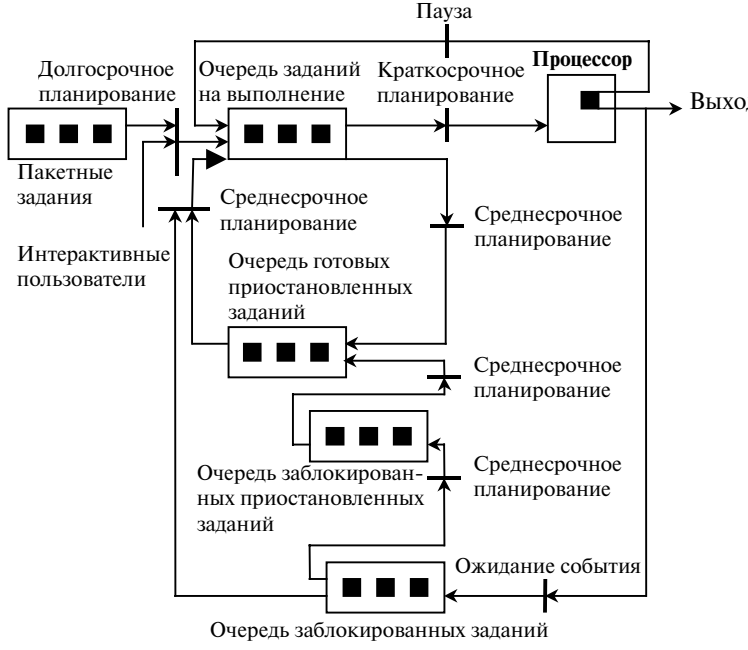
Рис. 3.7.2, А иллюстрирует изменение состояний объекта процесс, в результате выполнения метода планирования, что показывается черточкой, которая соединяется с объектом входной или выходной дугой.

Рис. 3.7.2, В иллюстрирует управление при планировании очередями объектов процессов с целью минимизации задержек и оптимизации производительности системы.

Долгосрочное планирование определяет степень многозадачности, допуская программу к выполнению. Допущенная программа (задание) становится процессом, который добавляется в очередь для краткосрочного планирования или для среднесрочного планирования.



А. Изменение состояний объекта процесс и типы планирования



В. Очередь объектов к процессору и типы планирования

Рис. 3.7.2. Планирование и изменение состояния объекта процесс

В пакетной части операционной системы общего назначения задание направляется на диск и хранится в очереди пакетных заданий,

а долгосрочный планировщик по возможности создает процессы для заданий из очереди, определяя:

- способна ли система работать с дополнительным процессом, что определяется желаемым уровнем многозадачности. Чем больше процессов будет создано, тем меньший процент времени будет тратиться на выполнение каждого из них (поскольку в борьбе за одно и то же время конкурирует большее количество процессов). Долгосрочный планировщик может ограничить степень многозадачности для обеспечения удовлетворительного уровня обслуживания текущего множества процессов. Кроме того, долгосрочный планировщик может быть вызван, когда относительное время простоя процессора превышает некоторый предопределенный порог;
- какое именно задание следует превратить в процесс(ы), здесь возможно использование нескольких критериев: приоритет, ожидание времени выполнения и требования к работе устройств ввода-вывода.

Среднесрочное планирование является частью системы свопинга. Обычно решение о загрузке процесса в память принимается в зависимости от степени многозадачности. В системе с отсутствием виртуальной памяти среднесрочное планирование также тесно связано с вопросами управления памятью. Таким образом, решение о загрузке процесса в память должно учитывать требования к памяти выгружаемого процесса.

Краткосрочное планирование выполняется чаще всего, определяя, какой именно процесс будет выполняться следующим. Краткосрочный планировщик (диспетчер) вызывается при наступлении события, которое может приостановить текущий процесс или предоставить возможность прекратить выполнение данного процесса в пользу другого. Например, прерывание таймера, прерывание ввода-вывода, вызовы операционной системы, сигналы.

Методы планирования

Разработка стратегии планирования представляет собой поиск компромисса среди противоречивых требований: относительный вес каждого из критериев планирования определяется предназначением разрабатываемой операционной системы. В табл. 3.7.1 приведены пользовательские и системные критерии планирования.

Функция выбора (selection function) определяет, какой из готовых к выполнению процессов будет выбран следующим. Функция может быть основана на приоритете, требованиям к ресурсам или характеристиках выполнения процессов.

В этом случае имеют значение три величины:

w — время, затраченное к этому моменту системой (ожидание и выполнение);

e — время, затраченное к этому моменту на выполнение;

s — общее время обслуживания, требующееся процессу, включая e (обычно оценивается и задается пользователем).

Таблица 3.7.1

Критерии планирования

Критерий	Пользовательские, связанные с производительностью
Время оборота	Интервал времени между подачей процесса и его завершением. Включает время выполнения, время, затраченное на ожидание ресурсов, в том числе для процессора. Критерий применяется для пакетных заданий
Время отклика	Время в интерактивной системе, истекшее между подачей запроса и началом получения ответа на него. Стратегия планирования должна стремиться сократить время отклика и сохранять его в заданных приделах, при максимизации количества интерактивных пользователей
Предельный срок	Для каждого процесса указывается предельный срок завершения. При планировании ему подчиняются все прочие цели, для максимизации количества процессов завершающихся в срок
Пользовательские, иные	
Предсказуемость	Задание должно выполняться примерно в одно и то же количество времени и с одной и той же стоимостью, независимо от загрузки системы. При нестабильной работе система должна дополнительно настраиваться
Системные, связанные с производительностью	
Пропускная способность	Цель стратегии планирования максимизировать — количество процессов, завершающихся за единицу времени, что является мерой количества выполненной системой работы. Очевидно, что эта величина зависит от средней продолжительности процесса; однако на нее влияет и используемая стратегия планирования
Использование процессора	Использование процессора представляет собой процент времени, в течение которого процессор оказывается занят
Системные, иные	
Беспристрастность	При отсутствии дополнительных указаний от пользователя или системы все процессы должны рассматриваться как равнозначные и ни один процесс не должен подвергаться голоданию
Приоритеты	Стратегия планирования отдает предпочтение процессам с более высоким приоритетом
Баланс ресурсов	Стратегия планирования должна поддерживать занятость системных ресурсов. Предпочтение отдается процессу, который недостаточно использует системные ресурсы

В зависимости от того, в какой момент выполняется функция выбора, подразделяют два режима решения и соответствующие режимам алгоритмы: невытесняющие (non-preemptive) и вытесняющие (preemptive) (рис. 3.7.3).

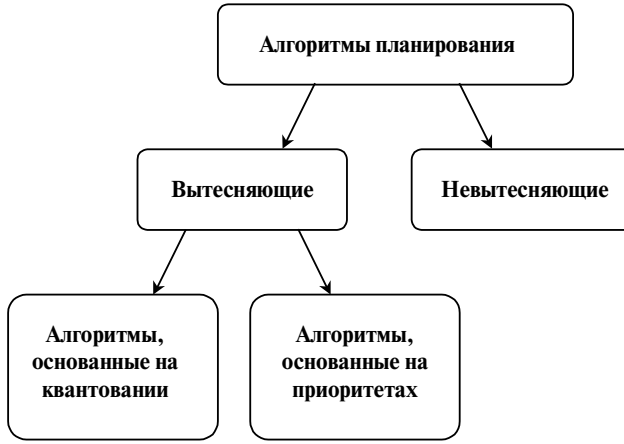


Рис. 3.7.3. Классификация алгоритмов планирования

Невытесняющие алгоритмы позволяют активному процессу выполняться, пока он сам, по собственной инициативе, не отдаст управление операционной системе для того, чтобы та выбрала из очереди другой готовый к выполнению поток.

Вытесняющие алгоритмы — способы планирования процессов, в которых решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается операционной системой, а не активной задачей.

Существенным преимуществом невытесняющих методов является более высокая скорость переключения с задачи на задачу.

Примером эффективного использования невытесняющей многозадачности является файл-сервер NetWare. Однако почти во всех современных операционных системах, ориентированных на высокопроизводительное выполнение приложений (UNIX, Windows NT, OS/2, VAX/VMS), реализована вытесняющая многозадачность.

Вытесняющие алгоритмы реализуют концепцию *квантования* и *приоритетного обслуживания*.

При *квантовании* каждому процессу поочередно для выполнения предоставляется ограниченный непрерывный период процессорного

времени — квант (time slicing). Смена активного процесса происходит, если: процесс завершился и покинул систему, произошла ошибка, процесс перешел в состояние ожидания, исчерпан квант процессорного времени, отведенный данному процессу.

Приоритет — это число, характеризующее степень привилегированности процесса при использовании ресурсов вычислительной машины, в частности процессорного времени: чем выше приоритет, тем выше привилегии, тем меньше времени будет проводить процесс в очередях.

На рис. 3.7.4 показана диаграмма использованных приоритетов. Вместо одной очереди готовых к исполнению процессов у нас имеется их множество, упорядоченное по убыванию приоритета: Q_0, Q_1, ..., Q_n, т. е.

Приоритет $[Q_i] >$ *Приоритет* $[Q_j]$ при $i < j$

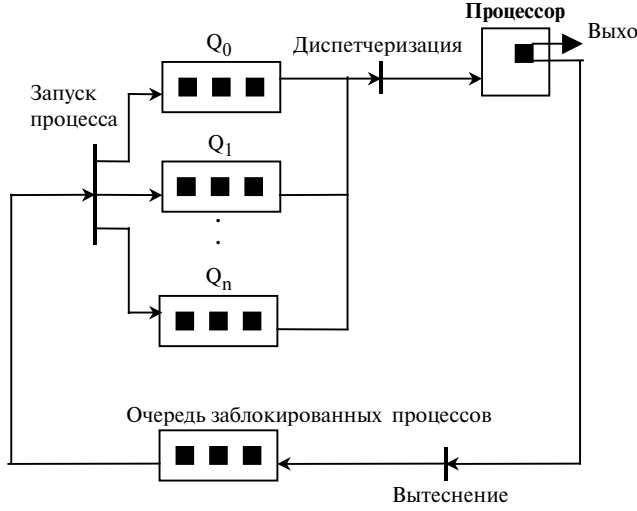


Рис. 3.7.4. Планирование с учетом приоритета процессов

При выборе процесса планировщик начинает с очереди процессов с наименьшим приоритетом Q_0. Если в очереди имеется один или несколько процессов, процесс для работы выбирается с использованием некоторой стратегии планирования. Если очередь Q_0 пуста, рассматривается очередь Q_1 и т. д.

Одна из основных проблем в такой чисто приоритетной схеме планирования в том, что процессы с низким приоритетом могут оказаться в состоянии голодания, если все время будут поступать новые процессы

с высоким приоритетом. Для решения этой проблемы приоритеты могут понижаться.

От того, какие приоритеты назначены процессам, существенно зависит эффективность работы всей системы. Рис. 3.7.5 иллюстрирует схему назначения приоритетов в ОС Windows.

Во многих операционных системах алгоритмы планирования построены с использованием как квантования, так и приоритетов. Например, в основе планирования лежит квантование, но величина кванта и/или порядок выбора процесса из очереди готовых определяется приоритетами процессов.

Рис. 3.7.5. Схема назначения приоритетов в Windows

В Windows с течением времени приоритет потока с переменным приоритетом может отклоняться от базового значения. ОС может повышать приоритет потока, в этом случае приоритет называется динамическим, если квант использовался не полностью, или понижать приоритет при полностью использованном кванте.

Приоритет может выражаться целыми или дробными, положительным или отрицательным значением. Чем выше привилегии процесса, тем меньше времени он будет проводить в очередях. Приоритет может назначаться директивно администратором системы в зависимости от важности работы или внесенной платы либо вычисляться самой ОС по определенным правилам, он может оставаться фиксированным на протяжении всей жизни процесса либо изменяться во времени в соответствии с некоторым законом. В последнем случае приоритеты называются динамическими.

Существует две разновидности метода: на основе относительных и абсолютных приоритетов.

В обоих случаях выбор процесса на выполнение из очереди готовых осуществляется одинаково: выбирается процесс, имеющий наивысший приоритет. По-разному решается проблема определения момента смены активного процесса. В системах с относительными приоритетами активный процесс выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние ожидание (или же произойдет ошибка, или процесс завершится) — невытесняющий режим. В системах с абсолютными приоритетами выполнение активного процесса прерывается еще при одном условии: если в очереди готовых процессов появился процесс, приоритет которого выше приоритета активного процесса. В этом случае прерванный процесс переходит в состояние готовности — вытесняющий режим.

В табл. 3.7.2 приведены методы планирования, реализованные в контуре.

Для сравнения методов обслуживания применяют следующие характеристики: время входа T_e, время начала T_b, время обслуживания T_s, время завершения T_t, время оборота T_r и нормализованное полное время T_r / T_s.

Таблица 3.7.2

Характеристики методов контура

	Функция выбора	Режим	Влияние на процессы
1. FCFS	max[w]	Невытесняющий	Плохо сказывается на коротких процессах и процессах с интенсивным вводом-выводом
2. RR	Const	Вытесняющий	Беспристрастно
3. SPN	min[s]	Невытесняющий	Плохо сказывается на длинных процессах
4. SRT	min[s-e]	Вытесняющий	Плохо сказывается на длинных процессах
5. HRRN	max([w+s]/s)	Невытесняющий	Хороший баланс
6. DP	Priority[Qn]	Вытесняющий	Может привести к предпочтению процессов с интенсивным вводом-выводом

1. Первым поступил — первым обслужен (FCFS — first-come-first-served), также известна как «первым пришел — первым вышел» (FIFO — first-in-first-out). Этот метод реализован в контуре (см. класс Scheduler). Как только процесс становится готовым к выполнению, он присоединяется к очереди готовых процессов. При прекращении выполнения текущего процесса для выполнения выбирается процесс, который находится в очереди дольше других.

Недостатки. Процессы, ориентированные на работу с процессором, получают преимущества над процессами, ориентированными на работу с вводом-выводом. При выполнении текущего процесса готовые процессы (ориентированные на работу с вводом-выводом) быстро проходят через состояние выполнения и тут же оказываются заблокированными очередной операцией ввода-вывода. Если же в этот момент окажется заблокированным и процесс, ориентированный на использование процессора, то последний будет находиться в состоянии простоя, что ведет к неэффективному использованию как процессора, так и устройств ввода-вывода.

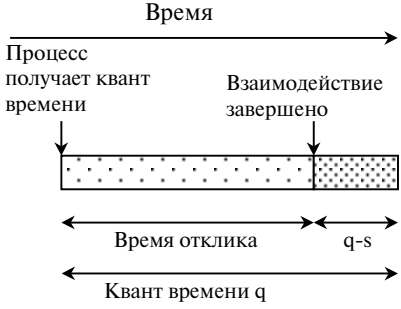
2. Круговое или карусельное планирование (RR — round robin) использует вытеснение на основе таймера для повышения эффективности работы с короткими процессами в методе FCFS.

Таймер генерирует прерывания через определенные интервалы времени. При каждом прерывании выполняющийся в настоящий момент процесс помещается в очередь готовых к выполнению процессов и начинает выполнять очередной процесс, выбираемый в соответствии со стратегией FCFS. Этот метод известен также как квантование времени, поскольку, перед тем как быть вытесненным, каждый процесс получает квант времени для выполнения. При этом принципиальным становится вопрос о продолжительности кванта времени. При малом кванте времени короткие процессы будут относительно быстро проходить через систему, но при этом возрастают накладные расходы, связанные с обработкой прерывания и выполнением функций планирования. Следует придерживаться правила: квант времени должен быть немного больше, чем время, требующееся для типичного полного обслуживания (рис. 3.7.6). Если квант оказывается меньшего размера, большинство процессов потребует как минимум два кванта времени. Когда квант времени превышает продолжительность самого длинного процесса, круговое планирование вырождается в планирование FCFS.

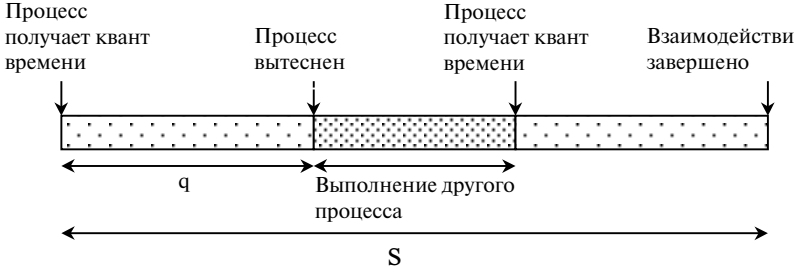
Кванты, выделяемые процессам, могут быть одинаковыми для всех процессов или различными. Кванты, выделяемые одному процессу, могут быть фиксированной величины или изменяться в разные периоды жизни процесса. Процессы, которые не полностью использовали выделенный им квант (например, из-за ухода на выполнение операций ввода-вывода), могут получить или не получить компенсацию в виде привилегий при последующем обслуживании.

В методе происходит смена активного процесса: если процесс завершился и покинул систему, произошла ошибка, процесс перешел в состояние блокирование (ожидание), исчерпан квант процессорного

времени, отведенный данному процессу. Процесс, который исчерпал свой квант, переводится в состояние готов и ожидает, когда ему будет предоставлен новый квант процессорного времени, а на выполнение в соответствии с определенным правилом выбирается новый процесс из очереди готовых.



А. Квант времени больше типичного взаимодействия



В. Квант времени меньше типичного взаимодействия

Рис. 3.7.6. Влияние продолжительности кванта времени на время отклика

Недостатки. Обычно у процесса с интенсивным вводом-выводом промежуток времени между двумя операциями с ввода-вывода, когда процесс использует процессор, меньше, чем у процесса, ориентированного на использование процессора. В результате возможна следующая ситуация: процесс с интенсивным вводом-выводом использует процессор в течение короткого промежутка времени и оказывается в заблокированном состоянии в ожидании завершения операции ввода-вывода. По завершении этой операции он вновь присоединяется к очереди готовых к выполнению процессов. Процесс, ориентированный на использование процессора, использует отпущенный ему квант времени полностью и немедленно возвращается в очередь готовых к выполне-

нию процессов. Таким образом, он получает больше процессорного времени, что приводит к снижению производительности процессов с интенсивным вводом-выводом, неэффективному использованию устройств ввода-вывода и увеличению времени отклика. Для решения этой проблемы вводится вспомогательная очередь для процессов, завершивших операции ввода-вывода, которой отдается преимущество при выборе процесса.

3. Выбор самого короткого процесса (SPN — shortest process next). Для выполнения выбирается процесс с наименьшим ожидаемым временем выполнения.

Трудность применения метода заключается в необходимости оценить время выполнения, требующееся каждому процессу. Для пакетных заданий время задается программистом: если оценка времени ниже, то система прекращает выполнение задания. Для часто выполняемых заданий может быть собрана статистическая информация. Для интерактивных процессов ОС может поддерживать во время выполнения средний «разрыв» для каждого процесса, вычисляемый по формуле

$$S_{n+1} = \frac{1}{n}\sum_{i=1}^{n} T_i,$$

где T_i — время работы процессора для i-го экземпляра данного процесса (общее время работы для пакетного задания, время разрыва при интерактивной работе); S_i — предсказанное значение для i-го экземпляра; S_1 — предсказанное значение для первого экземпляра (не вычисляется).

Для того чтобы избежать повторного вычисления всей суммы, уравнение можно записать следующим образом:

$$S_{n+1} = \frac{1}{n}T_n + \frac{n-1}{n}S_n.$$

Технология предсказания будущего значения на основе значений прошедших серий представляет собой взвешенное усреднение:

$$S_{n+1} = \alpha T_n + (1-\alpha)\,\alpha T_{n-1} + \ldots + (1-\alpha)^i\,\alpha T_{i-1} + \ldots (1-\alpha)^n S_1,$$

где α — постоянный множитель ($0 < \alpha < 1$), определяющий относительный вес последнего и предыдущих наблюдений. При использовании постоянного значения α, не зависящего от количества наблюдений, получаем ситуацию, когда рассматриваются все прошлые значения. Например, при $\alpha = 0.8$:

$$S_{n+1} = 0.8T_n + 0.16T_{n-1} + 0.032T_{n-2} + 0.0064T_{n-3} \ldots,$$

т. е. чем старее наблюдение, тем меньший вклад оно вносит в вычисляемое среднее значение. Чем больше α, тем больший вес имеет по-

следнее наблюдение, а при $\alpha < 0.5$ больший вес имеют более поздние наблюдения. При α близким к 1 позволяют методу быстро реагировать на любые измерения, но увеличивается реакция на случайные изменения от среднего значения наблюдения, что приводит к излишне резким изменениям вычислительного значения.

Недостатки. При стабильной работе коротких процессов длинные процессы могут голодать. Кроме того, применение метода нежелательно в системах с разделением времени и обработки транзакций из-за отсутствия вытеснения.

4. Метод наименьшего оставшегося времени (SRT — shortest remaining time) представляет собой вытесняющую версию метода SPN. Планировщик выбирает процесс с наименьшим ожидаемым временем до окончания процесса. При присоединении нового процесса к очереди готовых к исполнению процессов может оказаться, что его оставшееся время в действительности меньше, чем оставшееся время выполняемого в настоящий момент процесса.

По сравнению с методом FCFS нет больших перекосов в пользу длинных процессов. В отличие от метода RR здесь не генерируются дополнительные прерывания, что снижает дополнительные расходы. Однако увеличиваются накладные расходы из-за необходимости фиксировать и записывать время выполнения процессов.

Планировщик может применить вытеснение при готовности нового процесса и оценить время выполнения процесса.

Недостатки. Короткие процессы получают преимущества по сравнению с длинными процессами. Возможно, что длинные процессы могут голодать.

5. Наивысшее отношение отклика (HRRN — highest response ration next). Для каждого отдельного процесса показатель нормализованного времени оборота желательно минимизировать. Оценка может быть произведена на основе предыдущих выполнений или задана пользователем:

$$R = \frac{w+s}{s},$$

где R — отношение отклика; w — время, затрачиваемое процессом на ожидание; s — ожидаемое время обслуживания.

Если процесс будет немедленно диспетчеризован, его значение R будет равно нормализованному времени оборота 1.0.

При завершении или блокировании текущего процесса для выполнения из очереди готовых процессов выбирается тот, который имеет наибольшее значение R, что учитывает возраст процесса.

Короткие процессы получают преимущество по отношению к продолжительным (в силу меньшего знаменателя, увеличивающего отношение), однако и увеличение возраста процесса приводит к тому же результату, так что длинные процессы смогут конкурировать с короткими.

Как и для SRT и SPN в методе требуется оценка времени обслуживания для определения максимального значения *R*.

6. С динамическим (со снижением) приоритетом (DP — Dynamic Priority). При отсутствии информации о продолжительности процессов нет возможности использовать методы SPN, SRT или HRRN. Еще один путь предоставления преимущества коротким процессам — применение штрафных санкций к долго выполняющимся процессам, определяемым по затраченному ими времени.

Пусть выполняется вытесняющее по квантам времени планирование с использованием динамического механизма. При входе процесса в систему он помещается в очередь Q_0 (см. рис. 3.7.4). После первого выполнения и возвращения в состояние готовности процесс помещается в очередь Q_1. В дальнейшем при каждом вытеснении этого процесса он вносится в очередь с все меньшим приоритетом. Быстро выполняющиеся короткие процессы не могут далеко зайти в иерархии приоритетов, в то время как длинные процессы постепенно теряют свой приоритет. Таким образом, новые короткие процессы получают преимущество в выполнении над старыми длинными процессами. В рамках каждой очереди для выбора процесса используется стратегия FCFS. По достижении очереди с наиболее низким приоритетом процесс уже не покидает ее, всякий раз после вытеснения попадая в нее вновь (очередь обрабатывается циклическим методом). Такой подход известен как многоуровневый возврат (multilevel feedback), так как при блокировании или вытеснении процесса осуществляется его возврат на очередной уровень приоритетности. Имеется несколько разновидностей метода.

Недостатки. Если вытеснение выполняется, как в методе RR через периодические интервалы, например 1, то время оборота длинных процессов резко растягивается. В системе с частым запуском новых процессов в связи с этим вполне вероятно появление голодания. Для уменьшения отрицательного эффекта мы можем использовать разное время вытеснения для процессов из разных очередей: процесс из очереди Q_0 выполняется в течение одной единицы времени и вытесняется; процесс из очереди Q_1 выполняется в течение двух единиц времени и т. д. — процесс из очереди Q_i выполняется до вытеснения в течение 2^i единиц времени.

Для алгоритмов статистического планирования используется категория Statistic, которая служит для наблюдений и реализует методы статистики.

Таблица 3.7.3

Статистика (Statistic)

Statistic()	Конструктор класса
void setObservation(string user, int tExec, int tServ)	Добавить данные для наблюдения процесса по пользователю (user), где tExec — время выполнения, tServ — время обслуживания
double getTpredict(string user, StatTime time)	Взвешенное среднее от 0 < alfa < 1 предсказание, где StatTime — перчисляемый тип для получения по tExec или по tServ
double getTimeThreshold (StatTime time)	Порог срабатывания, с минимально предполагаемым временем в соответствии с параметром наблюдения и числом наблюдений

II. Во второй части требуется провести сравнительный анализ методов планирования, применительно к заданному варианту схемы, используя контур.

Требования к сравнительному анализу.

- В сравнительном анализе должны быть учтены все методы планирования.
- Должны быть продемонстрированы методы планирования для заданной схемы в соответствии с вариантами организации виртуальной и основной памяти.
- Для демонстрации организуйте дамп — распечатку содержимого памяти (см. main контура лабораторная работа 7).

Методические указания.

- За единицу времени в контуре принят такт выполнения инструкции.
- При сравнении методов составляется табл. 3.7.4, которая заполняется в соответствии с графами.

Таблица 3.7.4

Сравнение методов планирования

Метод	Процесс	P_1	P_2
	Время входа T_e Время начала T_b Время обслуживания T_s		
1. FCFS	Время завершения T_t Время оборота T_r Нормализованное время T_r / T_s		
2. RR	...		

Продолжение табл. 3.7.4

3. SPN	...		
4. SRT	...		
5. HRRN	...		
6. DP	...		

Лабораторная работа № 8

Лабораторная работа состоит из двух частей: изучения методов многопроцессорного планирования и разработки графика отношения пропускной способности стратегии планирования, при наличии одного и более процессоров, для заданной схемы. Определение характеристики зернистости синхронизации.

I. В первой части требуется изучить: виды многопроцессорных систем, характеристику зернистости синхронизации, методы планирования процессов в многопроцессорных ОС и категории контура, поддерживающие многопроцессорное планирование: назначение процессоров с разделением загрузки.

Многопроцессорные системы подразделяются на следующие виды:

Кластеры или слабосвязанные системы. Состоят из набора относительно автономных систем: каждый процессор имеет собственную основную память и каналы ввода-вывода.

Функционально специализированные процессоры. Например, процессор ввода-вывода, который предоставляет свои услуги ведущему процессору.

Сильносвязанные системы. Множество процессоров совместно используют общую основную память и управляются общей ОС.

Характеристикой многопроцессорной системы является зернистость синхронизации (synchronization granularity) — характеризует частоту синхронизации между процессорами в системе. Различают: независимые процессы, большую (2000–1М), очень большую (200–2000), среднюю (20–200) и малую (<20) зернистость.

Независимые процессы — между процессорами нет явной синхронизации. Если пользователи совместно используют файлы или другую информацию, то индивидуальные системы объединяются в распределенную систему. По многим параметрам с точки зрения стоимости многопроцессорная система по сравнению с распределенной получается более эффективной, позволяя, например, сэкономить на дисках.

Большая и очень большая зернистость — между процессами наблюдается весьма редкая синхронизация. Такую ситуацию трактуют как множество параллельно выполняющихся процессов, которые работают в многозадачном режиме на одном процессоре с минимальным измене-

нием программного обеспечения и могут поддерживаться многопроцессорной системой. При очень редком взаимодействии процессов целесообразно использовать распределенную систему, так как при более частом взаимодействии расходы на пересылку информации могут привести к снижению производительности.

Средняя зернистость. Обычно между потоками одного приложения требуется более высокая степень координации и взаимодействия, чем между различными процессами, что и приводит к среднему уровню синхронизации. Для систем с независимыми процессами, большой и очень большой зернистости параллельность поддерживается аналогично многозадачной системе с одним процессором. Для средней зернистости планирование становится важной задачей. Правильное планирование одного потока может существенно повлиять на производительность в целом.

Малая зернистость — представляет собой сложное использование возможностей параллельных вычислений, чем использование потоков.

Методы планирования

При доступности множества процессоров правило максимальной загруженности процессора перестает быть первостепенным; вместо этого первостепенную важность приобретает обеспечение максимальной средней производительности приложений. Для этих целей применяют следующие методы планирования.

1. Назначение процессоров: архитектура ведущий/ведомый и равноправных процессоров.
2. Разделение загрузки.
3. Бригадное планирование ().
4. Динамическое планирование ().

1. **Назначение процессов процессорам** предполагает, что процессоры — это единый ресурс и все процессоры идентичны. Подразделяют статическое и динамическое назначение.

Если процесс назначается одному процессору постоянно, от момента активации и до его завершения, то для каждого процессора следует поддерживать отдельную краткосрочную очередь, что способствует уменьшению накладных расходов планирования процессов (поскольку назначение процесса процессору выполняется один раз). Кроме того, использование выделенных процессоров обеспечивает возможность применения стратегии, известной как групповое (group) или бригадное (gang) планирование.

Недостатком статического распределения является то, что, когда один процессор загружен работой, другой может простаивать. Для

предотвращения такой ситуации можно использовать очередь, общую для всех процессоров. Все процессы попадают в одну глобальную очередь и передаются для выполнения любому свободному процессору. Таким образом, в течение жизни процесс может в разное время выполняться на разных процессорах.

В сильно связанных системах с общей памятью информация контекста всех процессов доступна всем процессорам и, следовательно, стоимость планирования не зависит от того, какой из процессов оказывается выбранным.

Используются два основных подхода к назначению процессов процессорам: ведущий/ведомый и равноправные процессоры.

В архитектуре ведущий/ведомый функции ОС выполняются на одном специально выделенном процессоре; все остальные процессоры могут выполнять только пользовательские приложения. Ведущий процессор отвечает за планирование заданий. Когда активный процесс на ведомом процессоре требует определенного обслуживания (например, осуществляет вызов ввода-вывода), он должен послать запрос ведущему процессору и ожидать завершение сервиса. Такой подход требует небольших изменений в однопроцессорную многозадачную ОС. Один процессор управляет всей памятью и вводом-выводом. Однако сбой ведущего процессора приводит к неработоспособности всей системы в целом, а ведущий процессор превращается в узкое место системы, определяющее ее производительность в целом.

В архитектуре равноправных процессоров ОС может выполняться на любом из процессоров, и каждый процессор самостоятельно планирует свою работу, беря процессы для выполнения из общего пула. ОС должна гарантировать, что никакие два процесса не выберут одновременно один и тот же процесс для выполнения и что не будет потерь из очереди. Запросы к ресурсам должны синхронизироваться.

Планирование назначения также применяется при назначении потоков процессорам. Каждой программе на время ее выполнения выделяется количество процессоров, равное количеству потоков программы. По завершении работы процессоры возвращаются в общий пул для распределения другими программами.

Однако если поток приложения блокируется операцией ввода-вывода или необходимостью синхронизации с другим потоком, то процессор этого потока простаивает: многозадачность процессора в этом методе отсутствует. Устранение переключений процессов программы во время работы существенно повышает ее скорость работы.

2. **Разделение загрузки**. Процессы не назначаются конкретным процессорам. Поддерживается глобальная очередь готовых к выполнению

потоков, и каждый процессор в состоянии простоя выбирает поток из этой очереди. Термин *разделение загрузки* (load sharing) используется, чтобы отличить эту стратегию от схем со сбалансированной загрузкой, в которых работа распределяется на более постоянной основе.

Достоинства.

- Загрузка равномерно распределяется между процессорами, обеспечивая отсутствие простоя процессоров при наличии работы.
- Не требуется централизованный планировщик; когда процессор становится доступным, он сам выполняет подпрограмму планирования ОС для выбора очередного потока или процесса.
- Глобальная схема может быть организована с использованием любого метода однопроцессорного планирования.
- При поступлении в систему задания с несколькими потоками каждый из потоков помещается в конец разделяемой очереди. Когда процессор становится свободным, он выбирает очередной готовый к использованию поток и работает с ним до его завершения или блокирования.
- Выбор процесса с наименьшим числом потоков (и с вытеснением). Разделяемая очередь готовых к выполнению потоков организуется как приоритетная, причем наивысший приоритет у потоков тех заданий, у которых осталось наименьшее число не распланированных потоков. Задания с одинаковыми приоритетами упорядочиваются в соответствии с их поступлением в систему. Как и в случае FCFS, поток выполняется до его завершения и блокировки. Наивысший приоритет получают задания с наименьшим количеством нераспланированных потоков. Поступившие задания с меньшим, чем у выполняющегося количеством потоков, вытесняют потоки, выполняющиеся в настоящий момент.

Недостатки.

- Центральная очередь занимает область памяти, доступ к которой должен производиться с обеспечением взаимоисключений. С увеличением количества процессоров при их обращении к очереди может возникнуть узкое место.
- Вытесненные потоки могут продолжить выполнение на других процессорах, что снижает эффективность локальных кэшей процессора.
- Маловероятно, что все потоки одной программы получат одновременно доступ к процессорам, поэтому требуется координация между потоками программы, что снижает общую производительность.

3. **Бригадное планирование** (gang scheduling). Множество связанных процессов (потоков) распределяется для одновременной работы среди множества процессоров, по одному процессу (потоку) на процессор.

Достоинства.

- Если тесно связанные процессы выполняются параллельно, то блокирование, вызванное синхронизацией, снижается: кроме того, требуется меньше переключений процессов. Минимизация переключений в целом приводит к повышению производительности. Например, поток процесса выполняется и достигает точки синхронизации с другим потоком того же процесса. Если это второй поток не выполняется, но готов к выполнению, то первый поток приостанавливается в ожидании, когда некоторым другим процессором не будет выполнено переключение процессов для запуска, требующегося потока.
- Накладные расходы, связанные с одновременным планированием нескольких связанных потоков уменьшаются, поскольку одно решение влияет одновременно на целый ряд процессов и процессоров.

4. **Динамическое планирование.** Количество потоков процесса может изменяться во время работы.

Ответственность ОС за планирование ограничена распределением процессоров, в планирование вовлекается приложение. Когда задача требует один или несколько процессоров, происходит следующее: используются простаивающие процессы, если они есть, иначе выделяется единственный процессор, забираемый у задачи, которая в этот момент использует более одного процессора. Если запрос не может быть удовлетворен, то он остается невыполненным до тех пор, пока либо не появится свободный процессор, либо задача не снимет свой запрос. При освобождении одного или двух процессоров сканируется текущая очередь неудовлетворенных запросов на процессоры, задаче, не имеющей ни одного процессора, назначается процессор.

Категория контура MPDispatcher (MultiProcessor — MP) поддерживает многопроцессорное планирование, реализуется наследованием от класса Dispatcher и замещением функции dispatch() и ее составляющих.

II. Во второй части требуется разработать график отношение пропускной способности стратегии планирования, при наличии одного и более процессоров, для заданной схемы. Оценить зернистость.

Требования к графику:

- график должен учитывать отношение пропускной способности стратегий планирования, при наличии одного и более процессоров;

- для демонстрации организуйте дамп — распечатку содержимого памяти (см. main контура лабораторная работа 8).

Методические указания.

- Время обслуживания — Tserv, пропускная способность — количество процессов в системе выполненных за фиксированный интервал времени. Принципиальным при сравнении стратегий планирования остается функция выбора процесса для выполнения.

Таблица 3.8.1

Многопроцессорная диспетчеризация (MPDispatcher)

MPDispatcher (int SIZE_OF_VIRTUAL_MEMORY_IN_IMAGES, Scheduler * scheduler,MMU * mmu, int MAX_PROCESSOR): Dispatcher(SIZE_OF_VIRTUAL_MEMORY_IN_IMAGES, scheduler, mmu)	Конструктор класса, вызывает конструктор суперкласса
Void scheduleProcess(MMU * mmu, bool priority)	Сформировать задание, замещение scheduleProcess класса Dispatcher
Interrupt executeProcess (MMU * mmu)	Выполнить задания на процессорах
valarray <Scheduler *> vaScheduler; valarray <bool> vaStatus;	Массив планировщиков и их состояние

ГЛАВА 4

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К КУРСОВЫМ ПРОЕКТАМ

Целью курсового проекта является разработка и реализация сетевой ОС на основе контура. Для этого необходимо изучить следующие темы: 5. Архитектуры сетевых операционных систем. 6. Методы взаимодействия распределенных процессов. 7. Сетевая файловая служба и выполнить задания для курсового проекта.

Тема 5. Архитектуры сетевых операционных систем

В зависимости от того, как распределены функции между компьютерами сети, компьютеры могут выступать в трех разных ролях:

- серверный узел — компьютер занимается исключительно обслуживанием запросов других;
- клиентский узел — компьютер обращается с запросами к ресурсам другой машины;
- одноранговый узел — компьютер совмещает функции клиента и сервера.

Сеть, обеспечивающая взаимодействие компьютеров, может быть построена по одной из трех следующих схем:

1) на основе одноранговых узлов — одноранговая сеть;

2) на основе клиентов и серверов — сеть с выделенными серверами;

3) сеть, включающая узлы всех типов — гибридная сеть.

В одноранговых сетях все компьютеры равны в возможностях доступа к ресурсам друг друга (рис. 4.1). Одноранговые сети состоят только из одноранговых узлов. При этом все компьютеры в сети имеют потенциально равные возможности. Одноранговые ОС включают как серверные, так и клиентские компоненты сетевых служб. Одноранговые сети проще в организации и эксплуатации, по этой схеме организуется работа в небольших сетях, в которых количество компьютеров не превышает 10–20.

В сетях с выделенными серверами используются специальные варианты сетевых ОС, которые оптимизированы для работы в роли серверов и называются серверными ОС (рис. 4.2). Пользовательские компьютеры в этих сетях работают под управлением *клиентских ОС*.

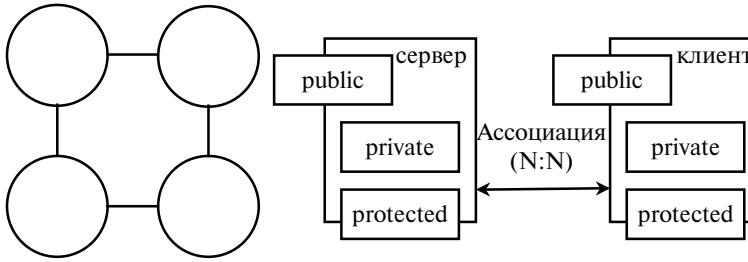


Рис. 4.1. Категория одноранговая сеть

На практике приложение обычно разделяют на две или три части. Наиболее распространенной является *двухзвенная архитектура*, распределяющая приложение между двумя компьютерами.

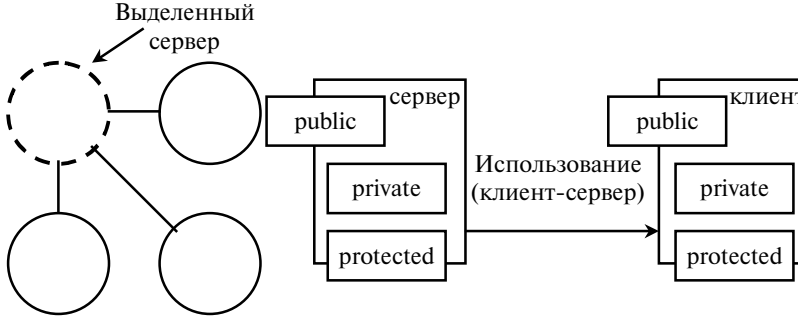


Рис. 4.2. Категория выделенный сервер

Рассмотрим сначала два крайних случая двухзвенной архитектуры, когда нагрузка в основном ложится на один узел — либо на центральный компьютер (централизованная архитектура), либо на клиентскую машину (файловый сервер).

Главным и очень серьезным недостатком централизованной архитектуры является ее недостаточная масштабируемость и отсутствие отказоустойчивости. Производительность центрального компьютера всегда будет ограничителем количества пользователей, работающих с данным приложением, а отказ центрального компьютера приводит к прекращению работы всех пользователей.

Другие варианты двухзвенной архитектуры более равномерно распределяют функции между клиентской и серверной частями системы. Наиболее часто используется архитектура, в которой на серверный компьютер возлагаются функции проведения внутренних операций базы

данных и файлов. Клиентский компьютер при этом выполняет все функции, специфические для данного приложения, а сервер — функции, реализация которых не зависит от специфики приложения, из-за чего эти функции могут быть оформлены в виде сетевых служб.

Трехзвенная архитектура позволяет лучше сбалансировать нагрузку на различные компьютеры в сети, а также способствует дальнейшей специализации серверов и средств разработки распределенных приложений. Примером трехзвенной архитектуры может служить такая организация приложения, при которой на клиентской машине выполняются средства представления и логика представления, а также поддерживается программный интерфейс для вызова частей приложения второго звена — промежуточного сервера. Промежуточный сервер называют в этом варианте сервером приложений, так как на нем выполняются прикладная логика и логика обработки данных.

Сервер баз данных, как и при двухзвенной архитектуре, выполняет функции двух последних слоев — операции внутри базы данных и файловые операции.

Централизованная реализация логики приложения решает проблему недостаточной вычислительной мощности клиентских компьютеров для сложных приложений, а также упрощает администрирование и сопровождение.

Монитор транзакций представляет собой популярный пример программного обеспечения, не входящего в состав сетевой ОС. Такой монитор управляет транзакциями с базой данных и поддерживает целостность распределенной базы данных.

Трехзвенные архитектуры часто применяются для централизованной реализации в сети некоторых общих для распределенных приложений функций, отличных от файлового сервиса и управления базами данных. Программные модули, выполняющие такие функции, относят к классу middleware — то есть промежуточному слою, располагающемуся между индивидуальной для каждого приложения логикой и сервером баз данных.

В гибридных сетях, обычно больших сетях, наряду с отношениями клиент-сервер сохраняется необходимость и в одноранговых связях (рис. 4.3).

Для серверных ОС характерны поддержка мощных аппаратных платформ, в том числе мультипроцессорных, широкий набор сетевых служб, поддержка большого числа одновременно выполняемых процессов и сетевых соединений, наличие развитых средств защиты и средств централизованного администрирования сети.

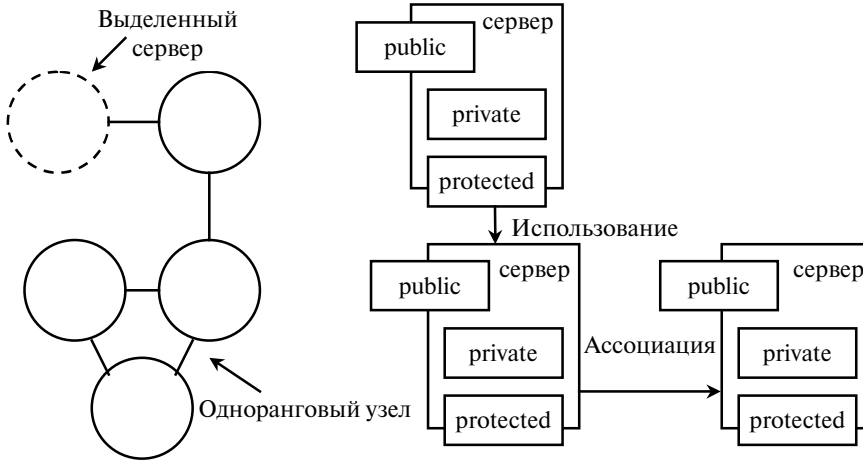


Рис. 4.3. Категория гибридная сеть

Клиентские ОС, в общем случае являясь более простыми, должны обеспечивать удобный пользовательский интерфейс и набор служб, позволяющий получать доступ к разнообразным сетевым ресурсам.

Тема 6. Методы взаимодействия распределенных процессов

Семафоры и мониторы не подходят для реализации обмена информацией между компьютерами. В распределенных системах межпроцессорное взаимодействие осуществляется следующими методами:

- передачей сообщений, например MPI (Mail Programming Interface);
- удаленным вызовом процедур RPC (Remote Procedure Call);
- сокетами (Sockets), например ОС UNIX.

Метод передачи сообщений использует два примитива: *send* и *receive*, которые скорее являются системными вызовами, чем структурными компонентами языка.

Примитивы *send* и *receive* реализуются в виде процедур

send (destination, Message &message);

receive (source, Message &message);

Категория сообщение (Message) — это блок информации, отформатированный процессом-отправителем таким образом, чтобы он был понятен процессу-получателю. Сообщение состоит из заголовка, обычно фиксированной длины, и набора данных определенного типа переменной длины. В заголовке содержатся следующие элементы.

- Адресное поле, состоящее из двух частей: адреса процесса-отправителя и адреса процесса-получателя. Адрес каждого процесса может, в свою очередь, иметь некоторую структуру, позволяющую найти нужный процесс в сети, состоящей из большого количества компьютеров.
- Идентификатор сообщения — последовательный номер. Используется для идентификации потерянных сообщений и дубликатов сообщений в случае отказа в сети.
- Структурированная информация, состоящая в общем случае из нескольких частей: поля типа данных, поля длины данных и поля значения данных.

При взаимодействии процессов на различных компьютерах, соединенных сетью, возникает опасность потери сообщения. Чтобы избежать потери сообщения, получатель отправляет отправителю *подтверждение* приема сообщения. Если отправитель не получает подтверждения через некоторое время, сообщение посылается снова.

Категория буфер (Buffer). При использовании синхронных примитивов используют буферизацию. Сообщение помещается в буфер компьютера-получателя, если в момент его прихода процесс-получатель не может обработать сообщение немедленно. Буфер может располагаться как в системной области памяти, так и в области памяти пользовательского процесса.

Для всех вариантов обмена сообщениями с помощью асинхронных примитивов необходима буферизация. Поскольку при асинхронном обмене процесс-отправитель может посылать сообщение всегда, когда ему это требуется, не дожидаясь подтверждения от процесса-получателя, для исключения потерь сообщений требуется буфер неограниченной длины.

При разработке категории следует предусмотреть примитив createBuffer(,,,) для создания буферов сообщений. Процесс должен использовать его перед тем, как отправлять или получать сообщения (send и receive). При создании буфера его размер может либо устанавливаться по умолчанию, либо выбираться прикладным процессом. Часто такой буфер носит название *порта* (port) или *почтового ящика* (mailbox).

При реализации категории буфер необходимо решить вопрос о том, что должна делать операционная система с поступившими сообщениями, для которых буфер не создан. Один из вариантов — просто отказаться от сообщения в расчете на то, что отправитель после тайм-аута передаст сообщение повторно и к этому времени получатель уже создаст буфер. Второй подход заключается в том, чтобы хранить хотя бы некото-

рое время поступающие сообщения в ядре получателя в расчете на то, что вскоре будет выполнен соответствующий примитив createBuffer(,,,). Каждый раз, когда поступает такое «неожидаемое» сообщение, включается таймер. Если заданный временной интервал истекает раньше, чем происходит создание буфера, то сообщение теряется.

Для решения проблемы, связанной с потерей сообщения, применяются три подхода.

Первый подход доставки сообщений называют *дейтаграммным* (datagram) — система не берет на себя никаких обязательств по поводу доставки сообщений. Взаимодействия реализует прикладной программист.

Во втором методе ядро принимающей машины посылает квитанцию-подтверждение ядру отправляющей машины на каждое сообщение или на группу последовательных сообщений. Посылающее ядро разблокирует пользовательский процесс только после получения такого подтверждения. Обработкой подтверждений занимается подсистема обмена сообщениями ОС, ни процесс-отправитель, ни процесс-получатель их не видят.

Третий метод заключается в использовании ответа в качестве подтверждения в тех системах, в которых запрос всегда сопровождается ответом, что характерно для клиент-серверных служб. В этом случае служебные сообщения-подтверждения не используются, так как в их роли выступают пользовательские сообщения-ответы. Процесс-отправитель остается заблокированным до получения ответа. Если же ответа нет слишком долго, то после истечения тайм-аута ОС отправителя повторно посылает запрос.

Для надежной и упорядоченной доставки требуется использовать обмен с предварительным установлением соединения. На стадии установления соединения стороны обмениваются начальными номерами сообщений, чтобы можно было в процессе обмена отслеживать как факт доставки отдельных сообщений последовательности, так и упорядочивать их.

Для реализации примитивов с различной степенью надежности передачи сообщений система обмена сообщениями ОС использует различные коммуникационные протоколы: TCP, UDP, SPX, IPX.

Подсистема передачи сообщений, называемая также транспортной подсистемой, обычно организуется как семиуровневая модель взаимодействия открытых систем (Open System Interconnection, OSI). Таким образом, в процесс выполнения примитивов send и receive вовлекаются средства всех нижележащих коммуникационных протоколов.

Метод сокетов — socket(...), например ОС UNIX. Для реализации метода должна быть разработана категория **Socket** (гнездо) — это точка, через которую сообщения уходят в сеть или принимаются из сети. Атрибуты категории: символьное имя (адрес), коммуникационный домен, например, в домене Интернета имя представлено парой (IP-адрес, порт).

Для обмена сообщениями категория сокетов должна реализовывать как системные вызовы следующие функции.

- Создание сокета: s = socket(domain. type, protocol).
  Процесс должен создать сокет перед началом его использования.
- Связывание сокета с адресом: bind(s. addr. addrlen).
  Системный вызов bind связывает созданный сокет с его высокоуровневым именем либо с низкоуровневым адресом.
- Запрос на установление соединения с удаленным сокетом:
  connect(s, server_addr. server_addrlen).
  Системный вызов connect используется только в том случае, если предполагается передавать сообщения в потоковом режиме, который требует установления соединения.
- Ожидание запроса на установление соединения: listen(s. backlog).
  Системный вызов listen используется для организации режима ожидания сервером запросов на установление соединения.
- Принятие запроса на установление соединения:
  snew = accept(s, client_addr, client_addrlen).
  Системный вызов accept используется сервером для приема запроса на установление соединения, поступившего от системного вызова listen через сокет s от клиента с адресом cient_addr.
- Отправка сообщения по установленному соединению:
  write(s. message. msglen).
  Сообщение длиной msglen, хранящееся в буфере message, отправляется получателю, с которым предварительно соединен сокет s.
- Прием сообщения по установленному соединению:
  nbytes =read(snew. buffer, amount).
  Сообщение, поступившее через сокет snew, с которым предварительно соединен отправитель, принимается в буфер buffer размером amount.
- Отправка сообщения без установления соединения:
  sendto(s. message. receiver_address).
  Так как сообщение отправляется без предварительного установления соединения, то в каждом системном вызове sendto необходимо указывать адрес сокета получателя.

- Прием сообщения без установления соединения:
  amount — recvfromCs. message, sender_address).

Метод удаленного вызова процедур RPC (Remote Procedure Call) представляет собой надстройку над системой обмена сообщениями ОС. Характерными чертами являются:

- асимметричность — одна из взаимодействующих сторон является инициатором взаимодействия;
- синхронность — выполнение вызывающей процедуры блокируется с момента выдачи запроса и возобновляется только после возврата из вызываемой процедуры.

Так как вызывающая и вызываемая процедуры выполняются на разных машинах, то они имеют разные адресные пространства и это создает проблемы при передаче параметров и результатов, особенно если машины и их операционные системы не идентичны.

Категория RPC должна реализовывать в рамках единого процесса, выполнение вызывающей программы и вызываемой локальной процедуры в одной машине, но в реализации RPC должны участвовать как минимум два процесса — по одному в каждой машине.

Стабы должны генерироваться автоматически, при этом программист должен описать интерфейс между клиентом и сервером RPC, включающий набор процедур, выполняющих взаимосвязанные функции, например функции доступа к файлам, функции удаленной печати и т. п. Поэтому при вызове удаленной процедуры задается интерфейс и процедура, поддерживаемая этим интерфейсом. Часто интерфейс также называют сервером RPC.

Связывание (binding) — процедура, устанавливающая соответствие между клиентом и сервером RPC. Различают статическое и динамическое связывание:

- при статическом связывании задается имя или адрес сервера RPC в явной форме, в качестве аргумента клиентского стаба или программы-сервера;
- динамическое связывание требует изменения способа именования сервера, с помощью имени RPC-интерфейса, состоящего из двух частей: типа интерфейса (все его характеристики) и экземпляра интерфейса. Динамическое связывание иногда называют импортом/экспортом интерфейса: клиент импортирует интерфейс, а сервер его экспортирует.

Процесс обнаружения требуемого сервера в сети строится двумя способами:

1) широковещательным (распространение по сети серверами RPC имени своего интерфейса и адреса экземпляра);

2) использованием централизованного агента (предполагает наличие в сети сервера имен, который связывает тип интерфейса с адресом сервера, поддерживающего такой же интерфейс). Интерфейс импортируется или экспортируется. Он может также поддерживать аутентификацию клиента.

Недостатки динамического связывания — это дополнительные накладные расходы (временные затраты) на экспорт и импорт интерфейсов. Кроме того, в больших распределенных системах может стать узким местом агент связывания, и тогда необходимо использовать распределенную систему агентов, что можно сделать стандартным способом, используя распределенную справочную.

Категория RPC должна оперировать двумя типами сообщений: сообщениями-вызовами, с помощью которых клиент запрашивает у сервера выполнение определенной удаленной процедуры и передает ее аргументы; сообщениями-ответами, с помощью которых сервер возвращает результат работы удаленной процедуры клиенту.

Тема 7. Сетевая файловая служба

Сетевая служба — совокупность серверной и клиентской частей, предоставляющих пользователям сети набор услуг — сетевой сервис: доступ к конкретному типу ресурса компьютера через сеть. Для каждой службы определен тип сетевых ресурсов и/или способ доступа к этим ресурсам. Наиболее важными для пользователей сетевых ОС являются файловая служба и служба печати.

Сетевые службы могут быть: встроены в ОС, объединены в виде некоторой оболочки, поставляться в виде отдельного продукта.

1. Встроенные сетевые службы в ОС (рис. 4.4).

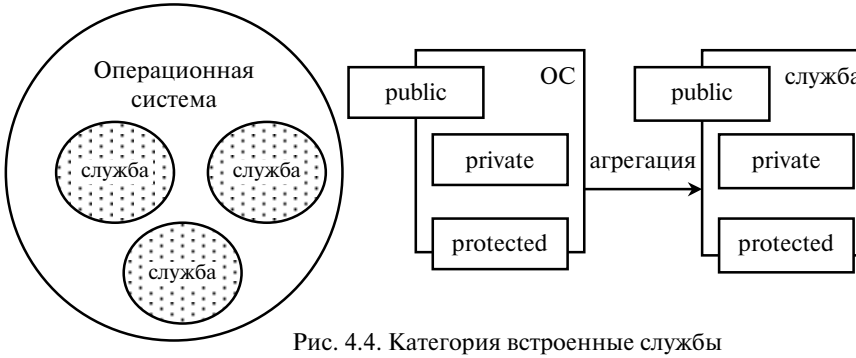


Рис. 4.4. Категория встроенные службы

2. Сетевые службы объединены в виде некоторого набора — оболочки (рис. 4.5).

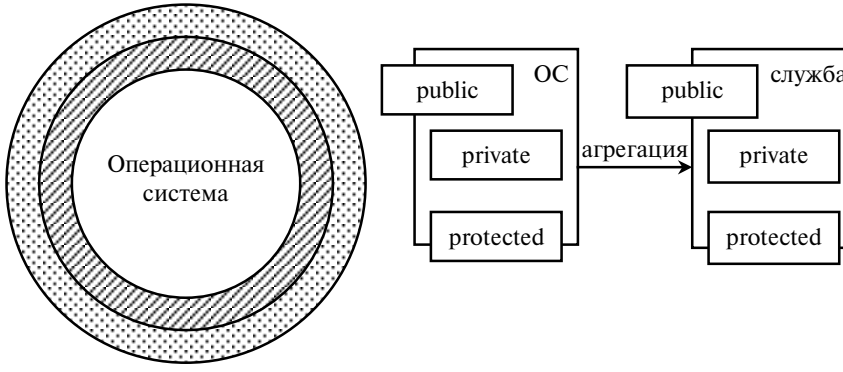


Рис. 4.5. Категория оболочки

3. Сетевые службы производятся и поставляются в виде отдельного продукта (рис. 4.6).

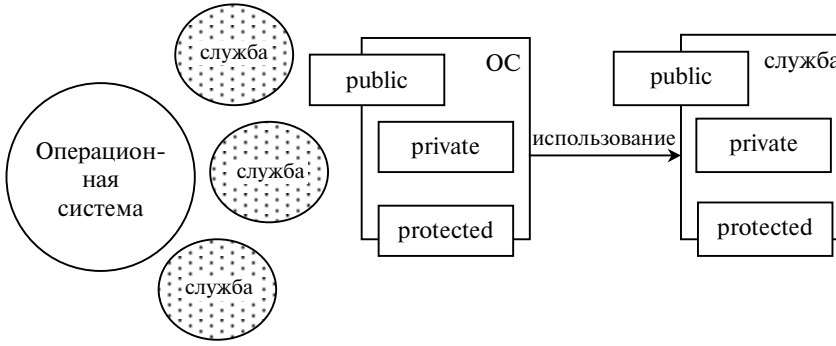


Рис. 4.6. Категория отдельные продукты

Файловый сервер — наиболее популярная сетевая служба сетевой файловой системы, которая лежит в основе многих распределенных приложений и некоторых других сетевых служб. Такая архитектура обладает хорошей масштабируемостью. Однако эта архитектура имеет и свои недостатки:

- во многих случаях резко возрастает сетевая нагрузка, что приводит к увеличению времени реакции приложения;

- компьютер клиента должен обладать высокой вычислительной мощностью, чтобы справляться с представлением данных, логикой приложения, логикой данных и поддержкой операций базы данных.

Распределенная файловая система поддерживается одним или более компьютерами, хранящими файлы. Эти компьютеры называют файловыми серверами. Файловые серверы отрабатывают запросы на чтение или запись файлов, поступающие от других компьютеров сети, которые в этом случае являются клиентами файловой службы. При этом данные монтируемых файловых систем физически никуда не перемещаются, оставаясь на серверах.

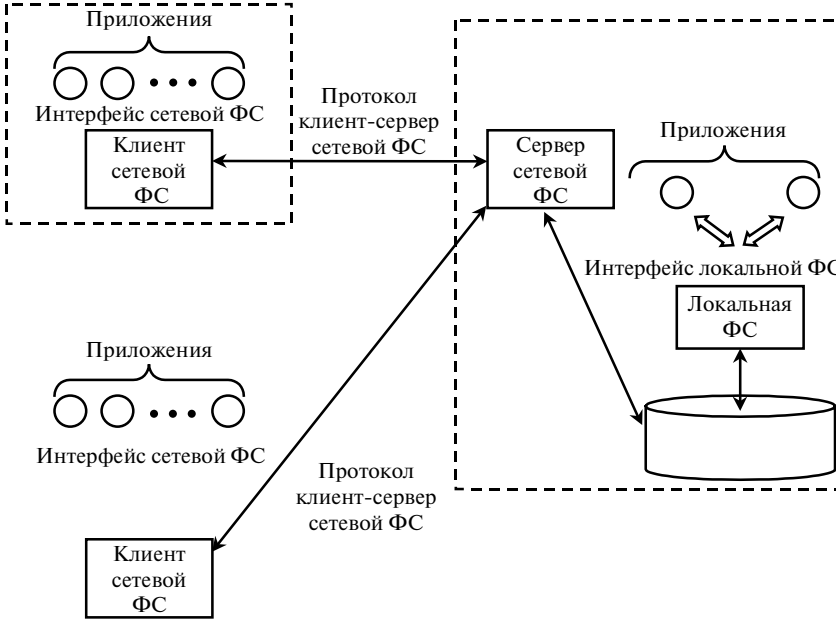


Рис. 4.7. Архитектура сетевой файловой системы

С программной точки зрения распределенная файловая система — это сетевая служба, имеющая типичную структуру. Файловая служба включает программы-серверы и программы-клиенты, взаимодействующие с помощью определенного протокола по сети между собой. В сети может одновременно работать несколько программных файловых серверов, каждый из которых предлагает различные файловые услуги.

Файловая служба в распределенных файловых системах имеет две функционально различные части: собственно файловую службу и службу каталогов файловой системы. Первая имеет дело с операциями над отдельными файлами, такими как чтение, запись или добавление, а вторая — с созданием каталогов и управлением ими, добавлением и удалением файлов из каталогов и т. п.

Категория файловая служба (ФС) должна включать (рис. 4.7):

- локальную файловую систему;
- интерфейс локальной файловой системы;
- сервер сетевой файловой системы;
- клиент сетевой файловой системы;
- интерфейс сетевой файловой системы;
- протокол клиент-сервер сетевой файловой системы.

Клиенты сетевой ФС — это программы, которые работают на многочисленных компьютерах, подключенных к сети. Эти программы обслуживают запросы приложений на доступ к файлам, хранящимся на удаленном компьютере.

Приложения обращаются к клиенту сетевой ФС, используя определенный программный интерфейс. Клиент и сервер сетевой файловой системы взаимодействуют друг с другом по сети по определенному протоколу.

Задания для курсового проекта

Задание 1. Разработайте категорию сетевой ОС (NetOS) как агрегацию всех разработанных в лабораторных работах категорий, категорий разработанных для архитектуры сети, категории взаимодействия процессов и файловых служб, а также категорий контура. Сборка должна быть произведена в соответствии с выбранной архитектурой ядра.

Для категории Диспетчер разработайте интерфейс с пользователем, который в графическом режиме должен выводить информацию о процессах. Используйте информацию, показывающую работу контура в лабораторных работах. Информацию следует представлять в виде графиков.

Задание 2. Разработайте конфигуратор для параметров сетевой ОС. Требования к конфигуратору. Конфигуратор размещается в файле config.xml. Параметры считываются, и по ним настраивается архитектура ОС, например, размер оперативной памяти, размер страницы, время прерывания таймера и т. д.

Задание 3. Процессы (P_1, P_2) заданной схемы должны быть рассмотрены как части одного распределенного (сетевого) приложения. Варианты:

1) разделение приложения на части, выполняющиеся на разных компьютерах сети;

2) выделение специализированных серверов в сети, на которых выполняются некоторые общие для приложения функции;

3) взаимодействия между частями приложений, работающих на разных компьютерах.

Задание 4. Курсовой проект должен содержать три тематических раздела:

1) разработка категории архитектуры сетевой ОС, варианты: клиентская, серверная, клиент-серверная;

2) разработка категории сетевой файловой службы, варианты: встроенная, объединенная оболочкой, отдельный продукт;

3) разработка категории взаимодействия распределенных процессов, варианты: передача сообщений, удаленный вызов процедур и сокеты.

Задание 5. Разработайте тестовые примеры. Для тестовых примеров используйте заданную схему и демонстрационные примеры контура. Пользователь должен иметь возможность наблюдать за работой сетевой ОС по планированию и диспетчеризации процессов.

Тестовые примеры должны показывать работу собранной ОС в сети. Информация, выводимая диспетчером, должна содержать:

1. Архитектуру сети.

2. Состояние процессов, идентификатор ID, пользователь, временные характеристики, метод планирования, диаграмму взаимодействия процессов.

3. Состояние протоколов.

4. Состояние виртуальной и основной памяти.

5. Трассировку процессов. Трассировка процессов включает основные события, которые так или иначе связаны с процессом. Здесь отображается информация о свопинге: считывание и запись процесса в виртуальную память, информация о передачи данных, сообщения об ошибках, например, передача данных процессом по несуществующему протоколу, или попытка создания процесса, когда нет свободной памяти и т. д.

Задание 6. Оформите и документируйте проделанную работу в соответствии с заданиями. При документировании должны использоваться диаграммы категорий, включающие взаимодействие и состояние процессов.

Диаграмма модулей проекта должна поясняться таблицей категорий, раскрывающих содержимое модулей.

Требования к оформлению

Лабораторные работы и курсовой проект оформляются отдельно. Лабораторные работы должны содержать тестовые примеры в соответствии с заданием. Курсовой проект должен содержать оглавление, постановку задачи и перечень выполненных заданий. В курсовом проекте требуется:

1) продемонстрировать работу сетевой ОС на примере заданной схемы;

2) продемонстрировать последствия изменения конфигурации сетевой ОС;

3) предоставить документацию по проекту и по использованию сетевой ОС.

Приложение А

Варианты схем компонент компьютера и программных компонент

Варианты схем необходимо представить в виде уровней микроядра (см. лабораторную работу 2).

Для компонент схемы приняты следующие обозначения:

- аппаратные компоненты: CPU — процессор, T — Timer (таймер), Lan — устройство передачи информации по сети, протоколов и файлов), Dev — device (устройство ввода-вывода);
- программные компоненты: A — architecture (архитектура ядра), D — dispatcher (диспетчер), P — process (процесс, например, P_1, P_2, ...), F — file (файл), K — kernel (ядро операционной системы), Pr — protocol (протокол), Dat — data (данные файла), L — Loader (загрузчик).

Устройства управляют передачей информацией между различными компонентами системы. В контуре они реализованы категориями, а также обработчиками прерываний для обращения к соответствующему прерыванию устройству.

Процессы относятся к уровню приложения и при выполнении используют тот ресурс, который указан на схеме входными или выходными стрелками. Например, процесс P_2 на схеме 1 передает информацию процессу P_1. Процесс P_1 записывает информацию в файл F_1, процесс P_2 считывает информацию из файла F_1. При записи и считывании информации программа должна содержать инструкцию прерывания, для обращения к соответствующему устройству, указанному на схеме. Управление приложением осуществляется ОС.

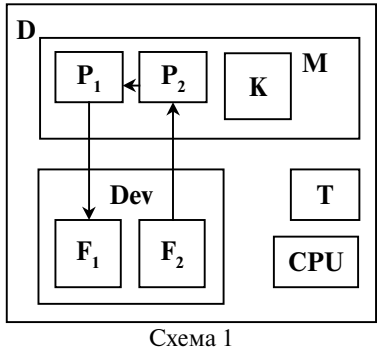


Схема 1

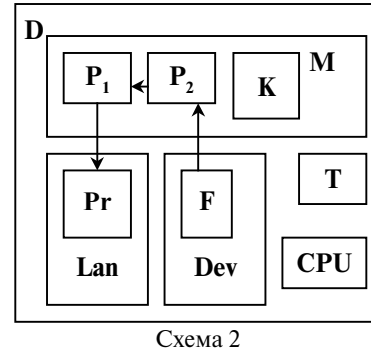


Схема 2

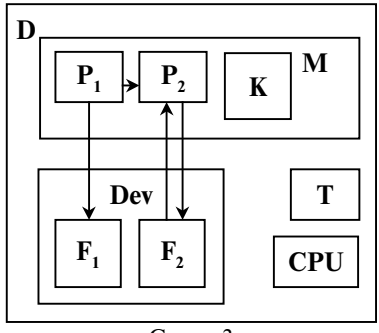


Схема 3

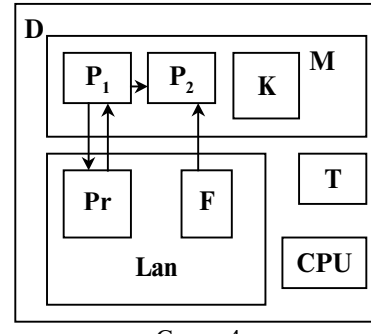


Схема 4

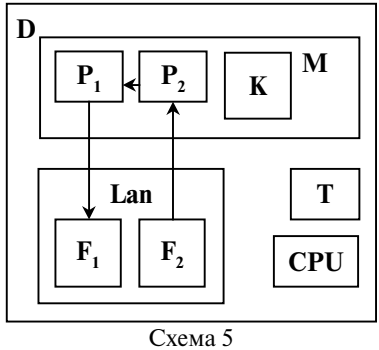


Схема 5

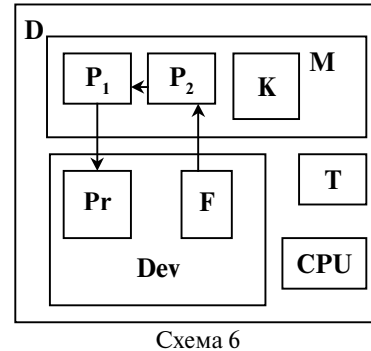


Схема 6

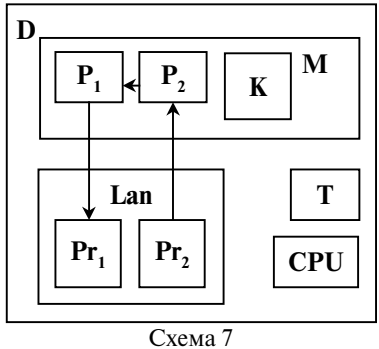


Схема 7

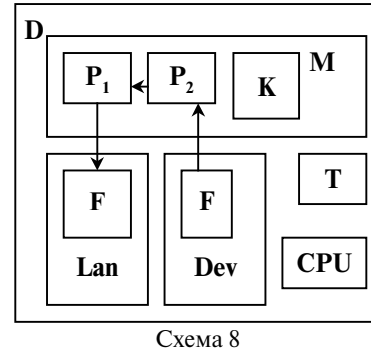


Схема 8

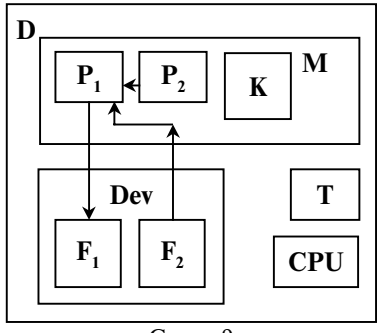


Схема 9

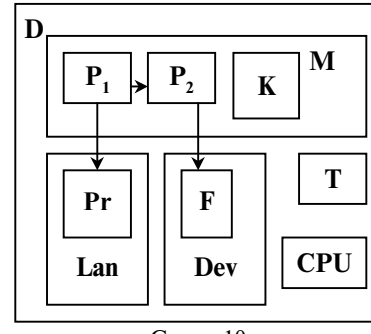


Схема 10

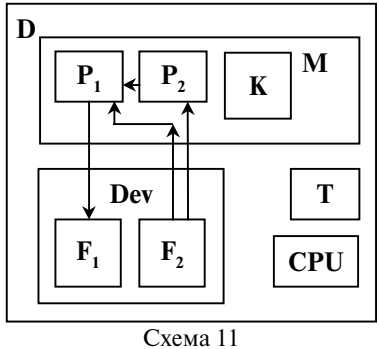


Схема 11

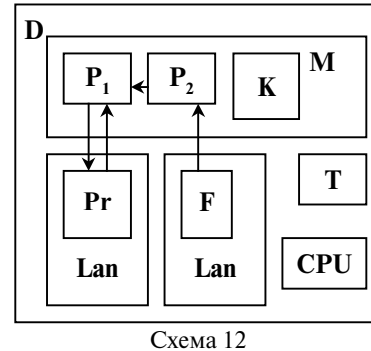


Схема 12

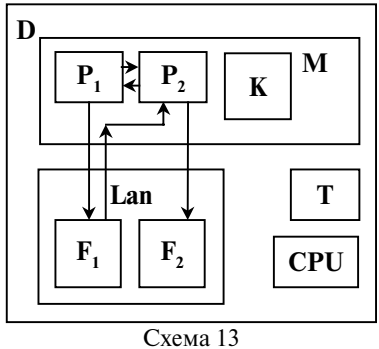


Схема 13

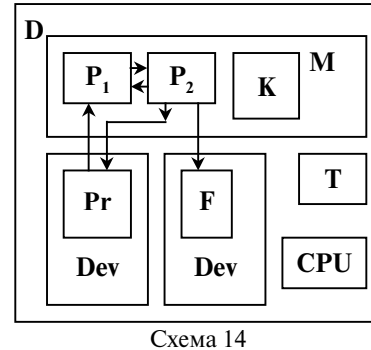


Схема 14

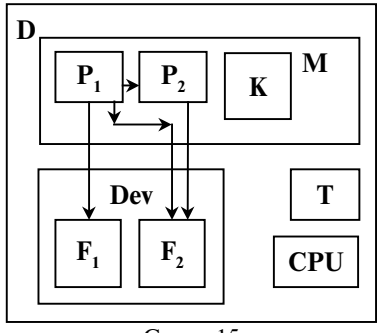


Схема 15

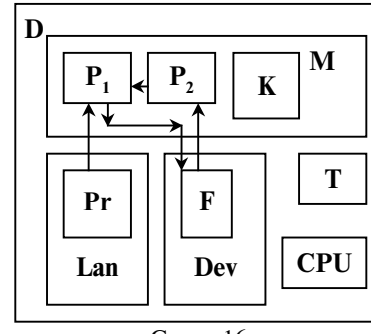


Схема 16

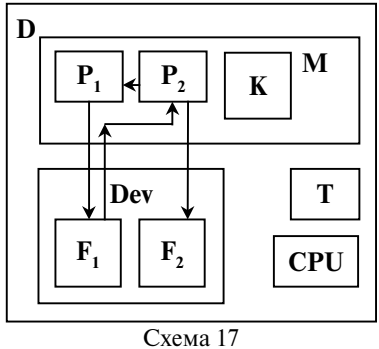


Схема 17

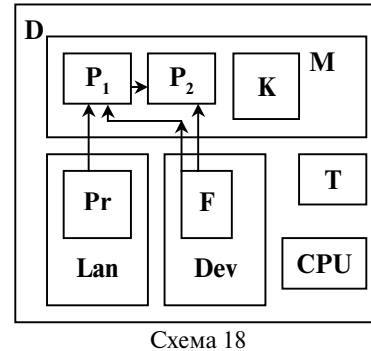


Схема 18

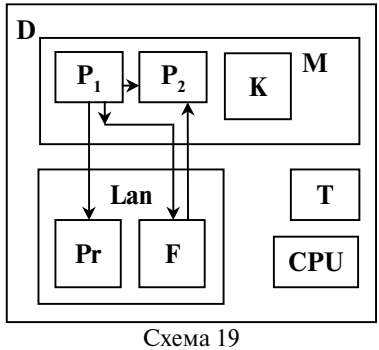


Схема 19

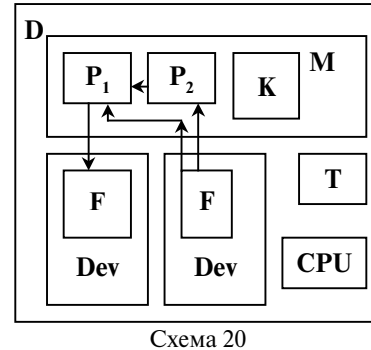


Схема 20

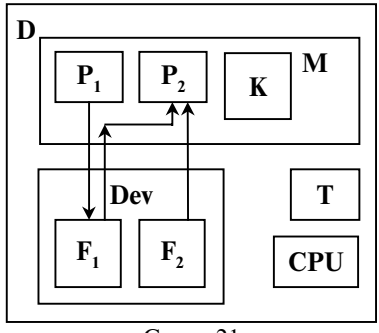


Схема 21

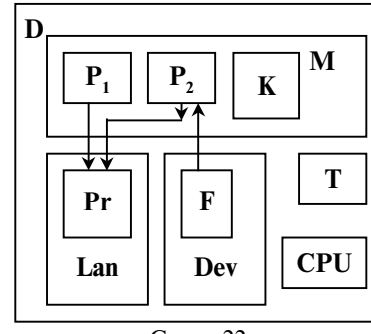


Схема 22

Приложение В

Контур для разработки сетевых операционных систем

Текст контура представлен в соответствии с диаграммой модулей.

Перечень модулей контура:

Модуль **main.cpp** — главная программа, с. 112;
Модуль **ArchitectureCPU.h**, с. 130;
Модуль **CPU.H**, с. 136;
Модуль **Memory.h**, с. 139;
Модуль **Device.h**, с. 141;
Модуль **LAN.h**, с. 141;
Модуль **HANDLE.h**, с. 142;
Модуль **Process.h**, с. 144;
Модуль **Dispatcher.h** с. 148;
Модуль **Kernel.h** с. 155;
Модуль **MMU.H** с. 156;
Модуль **VirtualMemory.h** с. 159;
Модуль **Scheduler.h** с. 161;
Модуль **Statistic.h** с. 174;
Модуль **MPDispatcher.h** с. 176;

Модуль main.cpp

```
#include "stdafx.h"
#include <string>
#include <iostream>
#include "CPU.H"
#include "MMU.H"
#include "Memory.h"
```

```
#include "Scheduler.h"
#include "Process.h"
#include "Kernel.h"
#include "HANDLE.h"
#include "Dispatcher.h"
#include "MPDispatcher.h"
using namespace std;

int main() {
 int SIZE_OF_MEMORY_IN_BLOCKS = 100;  /* размер
                                        оперативной памяти в блоках */
 int MAX_PROCESS = 3;  // максимальное количество процессов

 Memory memory(SIZE_OF_MEMORY_IN_BLOCKS);
 MMU mmu(memory);
 CPU *cpu = new CPU(memory,MAX_PROCESS, new LAN, new Device);
 Interrupt interrupt = OK;

 /* планирование расписания, выполнение программ
    для однопроцессорных систем */
 Scheduler scheduler(cpu, MAX_PROCESS);
 Job* job = new Job[MAX_PROCESS]; /* моделирование
                                     многозадачного режима 3 */
 char ch = 'Y';

 /* демонстрация создания образа процесса
    при  создании объектов класса Code указывается точный
    размер кода программы в блоках, т. е. количество строк */
 Code * code, *code_1, *code_2 = NULL;
 HANDLE *handle_1, * handle_2; // для двух процессов
 int SIZE_OF_VIRTUAL_MEMORY_IN_IMAGES = MAX_PROCESS; /* размер
                                     виртуальной памяти в образах процесса */

 Dispatcher dispatcher(SIZE_OF_VIRTUAL_MEMORY_IN_IMAGES,
                       &scheduler,&mmu);
 Kernel kernel(&dispatcher);

while(ch == 'Y' || ch == 'N'  ){
 cout << endl;
 cout << "dialog mode: \n"
"1. Computer architecture. execute programm 1,2 . . . . . .enter 1\n"
"2. OS architecture. MultiTasking execute programm 1 and 2. .enter 2\n"
"3. Model process. Create Process for programm 1 and 2. . . . .enter 3\n"
"4. Synchronize process.CriticalSection for programm 1 and 2.enter 4\n"
```

```
"5. Main memory and Management Memory Unit  . . . . . enter 5\n"
"6. Virtual memory and process control block - PCB. . . . . enter 6\n"
"7. Scheduling for one processor system. Methods. . . . . enter 7\n"
"8. MultiProcessor scheduling. Methods. . . . . enter 8"<< endl;
 cin >> ch;

 switch(ch){
  case '1':
  cout << " execute programm 1: Y/N-any "<< endl;
  cin >> ch;
  if (ch == 'Y') {
/* Programm 1: вычисление арифметического выражения
  (10*2 + 3) / 2 – 3 и вывод результата на устройство (Dev) печати
  Деление и умножение в модели CPU определены только между
  регистрами. Программа в начале работы размещается по
  нулевому адресу */
  memory.setCmd(Mov, r1, 2);  /* Перемещение содержимого
                              второго операнда в первый (регистр) */
  cout << endl;
  cout << "dump SysReg & first block execution: Y/N-any "<< endl;
  cin >> ch;
  if (ch == 'Y') { /* присвоить адрес в PC выполняемого блока;
                      для программы по умолчанию выделен пул
                      регистров с индесом 0 */
   scheduler.getSysReg(0)->setState(PC,0);
   scheduler.DebugBlock(0,cpu); /* трассировка состояния
                                   регистров и блока, его выполнение */
  }
  memory.setCmd(Mov, r2, 10);
  memory.setCmd(Mul, r1, r2);  //  регистр * регистр
  memory.setCmd(Add, r1, 3);
  memory.setCmd(Mov, r2, 2);
  memory.setCmd(Div, r1, r2);
  memory.setCmd(Sub, r1, 3);
  memory.setCmd(Wmem,r1, 69); /* запись содержимого r1 в память
                                 по адресу 69 */
  memory.setCmd(Int, r1, Dev); /* обращение к устройству для
                                  печати содержимого r1   */
  memory.setCmd(Int, r1, Lan); // посыл содержимого r1 по сети
  memory.setCmd(Int, 0, Exit);  /* все программы должны
                                  заканчиваться прерыванием Exit */
  cout << endl;
  cout << " programm 1 in memory: dump memory Y/N-any "<< endl;
  cin >> ch;
  if (ch == 'Y')  memory.debugHeap(); /* просмотреть содержимое
                                         памяти после загрузки программы */
```

```
  cout << " dump SysReg & execute programm 1: Y/N-any "<< endl;
  cin >> ch;
  if (ch == 'Y') {
   cout << " begin P1 cpu.execute "<< endl;
   scheduler.DebugSysReg(0);  /* трассировка состояния регистров
                                 до выполнения */
   /* выполнение программы, расположенной по адресу 0
      идентификатор  процесса  id = 0 для P1 */
   interrupt = scheduler.execute(0,0,cpu);
   scheduler.DebugSysReg(0); /* трассировка состояния регистров
                                после выполнения  */
   cout << " end   P1 cpu.execute interrupt = "<< interrupt << endl;
   cout << endl;
  }
  cout << " clear memory & dump: Y/N-any "<< endl;
  cin >> ch;
  if (ch == 'Y') {
   memory.clearMemory();
   memory.debugHeap();
   scheduler.getSysReg(0)->clearSysReg(); // очистить регистры
  }
  } // end execute programm 1
  cout << " execute programm 2: Y/N-any "<< endl;
  cin >> ch;
  if (ch == 'Y') {
/* Programm 2: умножение двух чисел  15 * 2
  mmu.setAlloc(0); — запись на место программы в памяти,
  память не очищается, запись производится в свободную память */
  mmu.setAlloc(20);
  memory.setCmd(Mov, r1, 15);
  memory.setCmd(Mov, r2, 2);
  memory.setCmd(Mul, r1, r2);       // регистр * регистр
  memory.setCmd(Wmem,r1, 70);  /* запись содержимого r1 в память
                                  по адресу 70   */
  memory.setCmd(Rmem,r1, 70);  /* чтение содержимого r1 из
                                  памяти по адресу 70   */
  memory.setCmd(Int, r1, Dev);     /* обращение к устройству для
                                  печати содержимого r1 */
  memory.setCmd(Int, 0, Exit);

  cout << endl;
  cout << " programm 2 in memory: dump memory Y/-any "<< endl;
  cin >> ch;
  if (ch == 'Y')  memory.debugHeap();  /* просмотреть содержимое
                                          памяти после загрузки программы */
```

```
  cout << " SysReg & execute programm 2: Y/-any "<< endl;
  cin >> ch;
  if (ch == 'Y') {
   cout << " begin P2 cpu.execute "<< endl;
   scheduler.DebugSysReg(1); // трассировка состояния регистров
   /* выполнение программы, расположенной по адресу 20
      идентификатор процесса id = 1 для P2 */
   interrupt = scheduler.execute(20,1,cpu);

   scheduler.DebugSysReg(1); // трассировка состояния регистров
   cout << " end   P2 cpu.execute interrupt = "<< interrupt << endl;
   cout << endl;
  }
  cout << " clear memory & execute programm 2 - Error: Y/N-any "<< endl;
  cin >> ch;
  if (ch == 'Y') {
   memory.clearMemory();
   cout << " begin P2 cpu.execute "<< endl;
   interrupt = scheduler.execute(20,1,cpu);
   cout << " end   P2 cpu.execute interrupt = "<< interrupt << endl;
   cout << endl;
   scheduler.getSysReg(1)->clearSysReg(); // очистить регистры
  }
  } // end execute programm 2
  break;
 case '2':
  cout << endl;
  cout << " clear memory & load P1: addr=10 & P2 addr=40" << endl;
  memory.clearMemory();

  mmu.setAlloc(10);
  memory.setCmd(Mov, r1, 2);   /* перемещение содержимого
                                  второго операнда в первый (регистр) */
  memory.setCmd(Mov, r2, 10);
  memory.setCmd(Mul, r1, r2);  //  регистр * регистр
  memory.setCmd(Add, r1, 3);
  memory.setCmd(Mov, r2, 2);
  memory.setCmd(Div, r1, r2);
  memory.setCmd(Sub, r1, 3);
  memory.setCmd(Wmem,r1, 69); /* запись содержимого r1 в память
                                  по адресу 69   */
  memory.setCmd(Int, r1, Dev);    /* обращение к устройству для
                                  печати содержимого r1 */
  memory.setCmd(Int, r1, Lan);    // посыл содержимого r1 по сети
  memory.setCmd(Int, 0, Exit);    /* все программы должны
                                  заканчиваться прерыванием Exit */
```

```
mmu.setAlloc(40);
memory.setCmd(Mov, r1, 15);
memory.setCmd(Mov, r2, 2);
memory.setCmd(Mul, r1, r2);  // регистр * регистр
memory.setCmd(Wmem,r1, 70);  /* запись содержимого r1 в память
                                по адресу 70 */

memory.setCmd(Rmem,r1, 70);  /* чтение содержимого r1 из
                                памяти по адресу 70 */
memory.setCmd(Int, r1, Dev);  /* обращение к устройству для
                                 печати содержимого r1 */
memory.setCmd(Int, 0, Exit);

cout << " dump memory: Y/N-any "<< endl;
cin >> ch;
if (ch == 'Y')
  memory.debugHeap();

/* в job: первый элемент начальный адрес процесса в памяти
          второй элемент отведен под id процесса
          третий элемент — индикатор выполнения задания */
job[0].set(10,0,true);
job[1].set(40,1,true);

cout << " Multitasking execute p1,p2: Y/N-any "<< endl;
cin >> ch;
if (ch == 'Y') {
  cout << " begin P1,P2 cpu.execute "<< endl;
  /* многозадачное выполнение программы:
     адреса задаются в массиве job (работа). По прерыванию
     таймера программа выполняет одну инструкцию */
  scheduler.setSliceCPU(1);
  interrupt = scheduler.execute(job, NULL);
  cout << " end   P1,P2 cpu.execute interrupt = "<< interrupt << endl;
}
cout << " dump memory: Y/N-any "<< endl;
cin >> ch;
if (ch == 'Y')
  memory.debugHeap();

cout << " clear memory: Y/N-any "<< endl;
cin >> ch;
if (ch == 'Y'){
  memory.clearMemory();
  memory.debugHeap();
```

```
    scheduler.getSysReg(1)->clearSysReg(); // очистить регистры
   }
   break;
 case '3':
   cout << endl;
   cout << " create Process P1 and P2 from code: " << endl;
 /*1. для каждой программы должен создаваться свой объект
    2. код в примерах записывается с 0 адреса */
   code = new Code(11);
   code->setCmd(Mov, r1, 2);  /* перемещение содержимого второго
                                 операнда в первый (регистр) */
   code->setCmd(Mov, r2, 10);
   code->setCmd(Mul, r1, r2);  // регистр * регистр
   code->setCmd(Add, r1, 3);
   code->setCmd(Mov, r2, 2);
   code->setCmd(Div, r1, r2);
   code->setCmd(Sub, r1, 3);
   code->setCmd(Wmem,r1, 69);  /* запись содержимого r1 в память по
                                  адресу 69 */
   code->setCmd(Int, r1, Dev);    /* обращение к устройству для печати
                                  содержимого r1 */
   code->setCmd(Int, r1, Lan);    // посыл содержимого r1 по сети
   code->setCmd(Int, 0, Exit);

   cout << " programm 1 in code: dump code Y/N-any "<< endl;
   cin >> ch;
   if (ch == 'Y')  code->debugHeap(); /* просмотреть содержимое
                                         памяти, в которой программа */

   handle_1 = kernel.CreateProcess("P1",code);  // создать процесс

   code = new Code(7);
   code->setCmd(Mov, r1, 15);
   code->setCmd(Mov, r2, 2);
   code->setCmd(Mul, r1, r2);
   code->setCmd(Wmem,r1, 70);
   code->setCmd(Rmem,r1, 70);
   code->setCmd(Int, r1, Dev);
   code->setCmd(Int, 0, Exit);

   handle_2 = kernel.CreateProcess("P2",code);  // создать процесс

   kernel.DebugProcessImage(handle_2);  // последний

   cout << " dump queue NotRunning Y/N-any "<< endl;
```

```
 cin >> ch;
 if (ch == 'Y')
  dispatcher.DebugQueue(NotRunning);

 cout << " dispatcher queue & execute Y/N-any "<< endl;
 cin >> ch;
 dispatcher.dispatch(); /* изменяет состояние системы,
                           т. к. ядро не под управлением CPU
                           для модели из трех очередей 2 вызова */

 cout << " dump memory: Y/N-any "<< endl;
 cin >> ch;
 if (ch == 'Y')
  memory.debugHeap();

 cout << " dump queue Running Y/N-any "<< endl;
 cin >> ch;
 if (ch == 'Y')
  dispatcher.DebugQueue(Running);

 cout <<"dispatch queue & dump queue ExitProcess Y/N-any"<< endl;
 cin >> ch;
 dispatcher.dispatch(); // изменяет состояние системы
 if (ch == 'Y')
  dispatcher.DebugQueue(ExitProcess);

 cout << " TerminateProcess & dump ProcessImage Y/N-any "<< endl;
 cin >> ch;
 kernel.TerminateProcess(handle_1);  // завершить процесс
 kernel.TerminateProcess(handle_2);  // завершить процесс
 if (ch == 'Y') {
   kernel.DebugProcessImage(handle_1);  // проверка; виртуальная
   kernel.DebugProcessImage(handle_2);  // память освобождена
 }
 memory.clearMemory();
 break;

case '4':
  cout << endl;
  cout <<"create unsynchronized Process P1 and P2,"
        "execute,look Error:"<< endl;
  code_1 = new Code(3);
  code_1->setCmd(Mov, r1, 15);
  code_1->setCmd(Wmem,r1, 70);  /* запись содержимого r1 в память
                                   по адресу 70 */
```

```
  code_1->setCmd(Int, 0, Exit);

  code_2 = new Code(5);
  code_2->setCmd(Rmem,r1, 70);  /* чтение содержимого r1 из
                                   памяти по адресу 70 */
  code_2->setCmd(Mov, r2, 2);
  code_2->setCmd(Mul, r1, r2);   // регистр * регистр
  code_2->setCmd(Int, r1, Dev);  /* обращение к устройству для
                                   печати содержимого r1   */
  code_2->setCmd(Int, 0, Exit);

  handle_1 = kernel.CreateProcess("P1",code_1);  // создать процесс
  handle_2 = kernel.CreateProcess("P2",code_2);  // создать процесс

  dispatcher.dispatch(); // изменяет состояние системы 2 вызова

  handle_1->ProcessTime();
  handle_2->ProcessTime();

/* программы работают асинхронно, поэтому P1 не успевает
   осуществить  запись содержимого r1 в память по адресу 70,
   строка 2 программа P2 читает содержимое r1 из памяти по
   адресу 70, но память пуста */

cout << " Dump memory Y/N-any "<< endl;
cin >> ch;
if (ch == 'Y') {
 memory.debugHeap();
}
kernel.TerminateProcess(handle_1); // завершить процесс
kernel.TerminateProcess(handle_2); // завершить процесс
memory.clearMemory();

cout << endl;
cout << " create synchronized Process P1 and P2" << endl;
cout << " using CriticalSection Y/N-any" << endl;
cin >> ch;
if (ch == 'Y') {

  handle_1 = kernel.CreateProcess("P1",code_1);  // создать процесс
  handle_2 = kernel.CreateProcess("P2",code_2);  // создать процесс

  kernel.EnterCriticalSection(handle_1);
  kernel.EnterCriticalSection(handle_2);
```

```
  // выполнение программы P1
  cout << " --------- P1 dispatch ----------" <<endl;
  dispatcher.dispatch(); // изменяет состояние системы 1-й шаг
    kernel.DebugProcessImage(handle_1);  // состояние процессов
    kernel.DebugProcessImage(handle_2);  // состояние процессов

  kernel.LeaveCriticalSection(handle_1);
  // выпополнение программы P2
  cout << " --------- P2  dispatch ---------" <<endl;
  dispatcher.dispatch(); // изменяет состояние системы 2-й шаг
    kernel.DebugProcessImage(handle_2);  /* проверка: виртуальная
                                            память освобождена */
    memory.debugHeap();
  kernel.LeaveCriticalSection(handle_2);
  cout << " dispatch" <<endl;
    dispatcher.dispatch(); // изменяет состояние системы 3-й шаг

  handle_1->ProcessTime();
  handle_2->ProcessTime();

  kernel.TerminateProcess(handle_1); // завершить процесс
  kernel.TerminateProcess(handle_2); // завершить процесс
}
memory.clearMemory();
break;
case '5':
  cout << endl;
  cout <<"Management Memory Unit dump Y/N-any"<< endl;
  /* пример программы показывает выполнение основных
     функций MMU — Management Memory Unit */
  code_1 = new Code(3);
  code_1->setCmd(Mov, r1, 15);
  code_1->setCmd(Wmem,r1, 70); /* запись содержимого r1 в
                                  память по адресу 70 */
  code_1->setCmd(Int, 0, Exit);
  cin >> ch;
  if (ch == 'Y') {
    cout <<"1.Create process using Management Memory Unit"<< endl;
    handle_1 = kernel.CreateProcess("P1",code_1);  // создать процесс
    mmu.Debug(handle_1);  // dump MMU — Management Memory Unit
  cout << "2. Execute process" << endl;
    dispatcher.dispatch();  // изменяет состояние системы 2 вызова
    dispatcher.dispatch();
  cout << "3. Terminate process " << endl;
    kernel.TerminateProcess(handle_1);  // завершить процесс
```

```
    // dump Terminate
    memory.clearMemory();
   }
   cout <<"Main memory and MMU Y/N-any"<< endl;
   cin >> ch;
   if (ch == 'Y') {
   cout << "1. Create process, load in main memory, execute" << endl;
    handle_1 = kernel.CreateProcess("P1",code_1);  // создать процесс
    dispatcher.dispatch();  // изменяет состояние системы 2 вызова
    memory.debugHeap();   /* программа удаляется из памяти;
                                данные расположенные по адресу 70 */
    dispatcher.dispatch();
   cout << "2. Terminate process " << endl;
    kernel.TerminateProcess(handle_1);   // завершить процесс
   cout << "3. Clear all memory" << endl;
    memory.clearMemory();
    memory.debugHeap();
   }
   break;
  case '6':
   cout << endl;
   cout <<"Virtual memory  Y/N-any"<< endl;
   /* пример иллюстрирует выполнение основных функций
      Virtual memory и process control block — PCB, Process Image*/
   code_1 = new Code(3);
   code_1->setCmd(Mov, r1, 15);
   // запись содержимого r1 в память по адресу 70
   code_1->setCmd(Wmem,r1, 70);
   code_1->setCmd(Int, 0, Exit);
   cin >> ch;
   if (ch == 'Y') {
    // 1. состояние виртуальной памяти в образах
   cout << "1. Virtual Memory state "<<endl;
    dispatcher.DebugVirtualMemory();  /* структура виртуальной
                                          памяти */
    // 2. выделение виртуальной памяти в образах
   cout << "2. Create process " << endl;
    handle_1 = kernel.CreateProcess("P1",code_1); // создать процесс
    dispatcher.DebugVirtualMemory();
    // изменяет состояние системы и выполнение программы
   cout << "3. Process execute " << endl;
    dispatcher.dispatch();
    dispatcher.dispatch();
   cout << "4. Terminate process " << endl;
    kernel.TerminateProcess(handle_1); // завершить процесс
```

```
    // 3. освобождение виртуальной памяти в образах
    dispatcher.DebugVirtualMemory();
    // 4. очистить основную память
    memory.clearMemory();
   }
   cout <<endl;
   cout<<"dump PCB and process image Y/N-any"<< endl;
   cin >> ch;
   if (ch == 'Y') {
   cout << "1. Create process " << endl;
    handle_1 = kernel.CreateProcess("P1",code_1); // создать процесс
    // 1. состояние блока PCB
    dispatcher.DebugPCB(handle_1);
   cout << "2. Process execute " << endl;
    dispatcher.dispatch(); // изменяет состояние системы 2 вызова
    // 2. состояние блока PCB после выполнения программы.
    //  см. состояние PC
    dispatcher.DebugPCB(handle_1);
    dispatcher.dispatch();
   cout << "3. Terminate process " << endl;
    kernel.TerminateProcess(handle_1);  // завершить процесс
  // 3. состояние блока PCB после удаления программы из системы
    dispatcher.DebugPCB(handle_1);
    memory.clearMemory();
   }
   break;
  case '7':
  /* 1. для программы "P1" и "P2" создаются объекты процессы
     сравнение методов планирования
     Tenter — Время входа
     Tbegin — Время начала
     Tservice — Время обслуживания в очередях
     Texec — Время выполнения
     Tterminate — Время завершения
     Tround = Tservice + Texec  — Время оборота
     Tround/ Tservice — Нормализованное время */

   code_1 = new Code(11);
   // перемещение содержимого второго операнда в первый (регистр)
   code_1->setCmd(Mov, r1, 2);
   code_1->setCmd(Mov, r2, 10);
   code_1->setCmd(Mul, r1, r2);  // регистр * регистр
   code_1->setCmd(Add, r1, 3);
   code_1->setCmd(Mov, r2, 2);
   code_1->setCmd(Div, r1, r2);
```

```
code_1->setCmd(Sub, r1, 3);
code_1->setCmd(Wmem,r1, 69);  /* запись содержимого r1 в память
                                 по адресу 69 */
code_1->setCmd(Int, r1, Dev); /* обращение к устройству для
                                 печати содержимого r1 */
code_1->setCmd(Int, r1, Lan); // посыл содержимого r1 по сети
code_1->setCmd(Int, 0, Exit);

code_2 = new Code(7);
code_2->setCmd(Mov, r1, 15);
code_2->setCmd(Mov, r2, 2);
code_2->setCmd(Mul, r1, r2);
code_2->setCmd(Wmem,r1, 70);
code_2->setCmd(Rmem,r1, 70);
code_2->setCmd(Int, r1, Dev);
code_2->setCmd(Int, 0, Exit);

cout << endl;
cout << " scheduler FCFS Y/N-any "<< endl;
cin >> ch;
if (ch == 'Y') {
 Timer::setTime(); // устанавливаем таймер в нулевое значение
 handle_1 = kernel.CreateProcess("P1",code_1);  // создать процесс
 handle_2 = kernel.CreateProcess("P2",code_2);  // создать процесс
 dispatcher.dispatch(); /* изменяет состояние системы
                           для модели из трех очерелей 2 вызова */
 dispatcher.dispatch();
 handle_1->ProcessTime();
 handle_2->ProcessTime();

 kernel.TerminateProcess(handle_1);   // завершить процесс
 kernel.TerminateProcess(handle_2);   // завершить процесс
 /* очереди должны быть пусты после эксперимента
    scheduler.DebugSPNQueue(); */
 memory.clearMemory();
}
cout << endl;
cout << " scheduler RR Y/N-any "<< endl;
cin >> ch;
if (ch == 'Y') {
 int slice = 5;
 Timer::setTime(); // устанавливаем таймер в нулевое значение
 /* в зависимости от размера программы и квоты, изменяется
    число строк dispatcher.dispatch(); */
 dispatcher.setTimeSlice(slice);
```

```
  handle_1 = kernel.CreateProcess("P1",code_1);  // создать процесс
  handle_2 = kernel.CreateProcess("P2",code_2);  // создать процесс
  int disp = 4; /* подобрано по диаграмме slice и организации
                   очередей   */
  for (int i =0; i < disp; i++){
   dispatcher.dispatch(); // изменяет состояние системы
  }

  handle_1->ProcessTime();
  handle_2->ProcessTime();

  kernel.TerminateProcess(handle_1);   // завершить процесс
  kernel.TerminateProcess(handle_2);   // завершить процесс
  memory.clearMemory();
  /* очереди должны быть пусты после эксперимента
     scheduler.DebugSPNQueue(); */
  dispatcher.resetTimeSlice();
}
cout << endl;
cout << " scheduler SPN (shortest process next)  Y/N-any "<< endl;
cin >> ch;
if (ch == 'Y') {
  int numProcess = 5; /* количество объектов одного процесса
       для расчета его предсказуемого времени выполнения  */
  Timer::setTime(); // устанавливаем таймер в нулевое значение
  // включить регистрацию наблюдений
  dispatcher.setTpredict(TimeExecNotPreem);
  /* для невытесняющего метода
     alfa = 0.8; установлен взвешенный коэффициент  0 < alfa < 1
     в ShortestProcessNext(), открыт отладчик */
  for (int i =0; i < numProcess; i++) {
   cout << endl;
   cout << " ---------------------- experiment i =  " << i <<endl;
   handle_1 = kernel.CreateProcess("user_1",code_1);
   handle_2 = kernel.CreateProcess("user_2",code_2);
   dispatcher.dispatch(); // изменяет состояние системы
   dispatcher.dispatch();
   if (i < numProcess-1 ){ /* –1, для сохранения информации
                              о последнем эксперименте */
     kernel.TerminateProcess(handle_1); // завершить процесс
     kernel.TerminateProcess(handle_2); // завершить процесс
   }
   /* наблюдения сохраняются, только после завершения
       процесса для первого процесса нет наблюдений */
   memory.clearMemory();
  }
```

```
  handle_1->ProcessTime();
  handle_2->ProcessTime();
  kernel.TerminateProcess(handle_1); // завершить процесс
  kernel.TerminateProcess(handle_2); // завершить процесс

  //scheduler.DebugSPNQueue();
  dispatcher.resetTpredict(); /* выключить регистрацию
                                 наблюдений */
}
cout << endl;
cout << " scheduler SRT (shortest remaining time)  Y/N-any "<< endl;
cin >> ch;
if (ch == 'Y') { // представляет собой вытесняющую версию SPN
 int numProcess = 3; /* количество объектов одного процесса
                        для расчета его предсказуемого времени выполнения  */
 Timer::setTime(); // устанавливаем таймер в нулевое значение
 // включить регистрацию наблюдений процессов
 dispatcher.setTpredict(TimeExec);
 // и вытеснение
 for (int i =0; i < numProcess; i++) {
  cout << endl;
  cout << " ---------------------- experiment i =  " << i <<endl;
 // параметр 0 включает наименьшее оставшееся время процесса
  handle_1 = kernel.CreateProcess("user_1",code_1);
  handle_2 = kernel.CreateProcess("user_2",code_2);

  dispatcher.dispatch(); // изменяет состояние системы
  dispatcher.dispatch(); // вытеснение user_1 и выполнение
  dispatcher.dispatch(); /* длинный процесс не в swapOut, не
                            успевает */

  if (i < numProcess-1 ){/* –1, для сохранения информации
                            о последнем эксперименте */
   kernel.TerminateProcess(handle_1);  // завершить процесс
   kernel.TerminateProcess(handle_2);   // завершить процесс
  }
 /* наблюдения сохраняются, только после завершения процесса
    для первого экземпляра процесса нет наблюдений */
  memory.clearMemory();
 }
 handle_1->ProcessTime();
 handle_2->ProcessTime();

 kernel.TerminateProcess(handle_1);   // завершить процесс
```

```
 kernel.TerminateProcess(handle_2);   // завершить процесс
 //scheduler.DebugSPNQueue();
 dispatcher.resetTpredict(); /* выключить регистрацию
                                наблюдений */
}

cout << endl;
cout << " scheduler HRRN (hightst response ratio next)  Y/N-any "<< endl;
cin >> ch;
if (ch == 'Y') { // представляет собой вытесняющую версию SPN
 int numProcess = 3; /* количество объектов одного процесса
                        для расчета его предсказуемого времени выполнения  */
 Timer::setTime(); // устанавливаем таймер в нулевое значение
 // включить регистрацию наблюдений процессов
 dispatcher.setTpredict(TimeServ);
 // в ShortestProcessNext(), открыт отладчик
 for (int i =0; i < numProcess; i++) {
  cout << endl;
  cout << " ---------------------- experiment i =  " << i <<endl;
 // параметр 0, включает наименьшее оставшееся время процесса
  handle_1 = kernel.CreateProcess("user_1",code_1);
  handle_2 = kernel.CreateProcess("user_2",code_2);

  dispatcher.dispatch(); // изменяет состояние системы
  dispatcher.dispatch(); // вытеснение user_1 и выполнение
  dispatcher.dispatch(); // процесс swapOut
  if (i < numProcess-1 ){ /* –1, для сохранения информации
                             о последнем эксперименте */
    kernel.TerminateProcess(handle_1); // завершить процесс
    kernel.TerminateProcess(handle_2); // завершить процесс
  /* наблюдения сохраняются, только после завершения процесса
     для первого экземпляра процесса нет наблюдений */
  }
  memory.clearMemory();
 }
 handle_1->ProcessTime();
 handle_2->ProcessTime();

 kernel.TerminateProcess(handle_1);  // завершить процесс
 kernel.TerminateProcess(handle_2);  //
 // scheduler.DebugSPNQueue();
 dispatcher.resetTpredict(); /* выключить регистрацию
                                наблюдений */
}
```

```
  cout << endl;
  cout << " scheduler dynamic priority DP Y/N-any "<< endl;
  cin >> ch;
  if (ch == 'Y') {
   int slice = 5;
   /* priority 0..3 индекс массива. TimeSlice — значение элемента в
       массиве */
   int * prSlice = new int [3];
   prSlice[0] = 5;
   prSlice[1] = 3;
   prSlice[2] = 2;
   Timer::setTime(); // устанавливаем таймер в нулевое значение
  /* в зависимости от размера программы и TimeSlice изменяется
      число строк dispatcher.dispatch(); */
   dispatcher.setTimeSlice(3,prSlice);
    handle_1 = kernel.CreateProcess("P1",code_1);  // создать процесс
   dispatcher.dispatch(); // изменяет состояние системы
    handle_2 = kernel.CreateProcess("P2",code_2);  // создать процесс

   int disp = 5; // где
   for (int i =0; i < disp; i++){
    dispatcher.dispatch(); // изменяет состояние системы
   }

   handle_1->ProcessTime();
   handle_2->ProcessTime();

   kernel.TerminateProcess(handle_1);  // завершить процесс
   kernel.TerminateProcess(handle_2);  // завершить процесс
   memory.clearMemory();

   //scheduler.DebugSPNQueue();
   dispatcher.resetTimeSlice();
  }
  break;
 case '8':
/* 1. сравнение методов планирования многопроцессорной
   системы  для программ "P1" и "P2" создаются объекты процессы

    Tenter — Время входа
    Tbegin — Время начала
    Tservice — Время обслуживания в очередях
    Texec — Время выполнения
    Tterminate — Время завершения
    Tround = Tservice + Texec  — Время оборота
    Tround/ Tservice — Нормализованное время */
```

```
{ // block begin

   code = new Code(5);
   code->setCmd(Mov, r1, 15);
   code->setCmd(Mov, r2, 2);
   code->setCmd(Mul, r1, r2);
   code->setCmd(Int, r1, Dev);
   code->setCmd(Int, 0, Exit);

   cout << endl;
   cout << " Assignment of Processes to Processors and FCFS Y/N-any "<< endl;
 /* MultiProcessor scheduling (MP) — многопроцессорное
    планирование. Метод Global queue — планирование очередей */
int MAX_PROCESSOR = 4;  /* максимальное количество
                           процессоров */
Scheduler mpScheduler(cpu, MAX_PROCESS);
MPDispatcher mpDispatcher
                (SIZE_OF_VIRTUAL_MEMORY_IN_IMAGES,
                 &mpScheduler,&mmu,MAX_PROCESSOR);
   Kernel mpKernel(&mpDispatcher);
   valarray <HANDLE *> vaHANDLE (MAX_PROCESS);

   cin >> ch;
   if (ch == 'Y') {
    Timer::setTime(); // устанавливаем таймер в нулевое значение

    // 1. Состояние процессоров
    mpDispatcher.MPDebug();
    for(int i=0; i < MAX_PROCESS; i++) {
     vaHANDLE[i] = mpKernel.CreateProcess("P",code); /* создать
                                                        процесс */
    }
    mpDispatcher.dispatch(); // изменяет состояние системы
    mpDispatcher.dispatch(); // для модели из трех очерелей 2 вызова
    mpDispatcher.dispatch();

    for(int i=0; i < MAX_PROCESS; i++) {
     vaHANDLE[i]->ProcessTime(); // создать процесс
     mpKernel.TerminateProcess(vaHANDLE[i]); // завершить процесс
    }
    memory.clearMemory();
    // mpScheduler.DebugSPNQueue();
   }
} // block end
```

```
   break;

   default: ;
 } // end case

}// end while
   return 0;
}
```

Модуль ArchitectureCPU.h

```
#ifndef __ArchitectureCPU_H__
#define __ArchitectureCPU_H__

#include <iostream>
using namespace std;

enum Instruction {Mov,Add,Sub,Mul,Div,And,Or,Xor,Shlu,
  Shru,Shls,Shrs,Cmp,Je,Jne,Jge,Jgt,Jle,Jlt,Int,Pushw,
  Pushc,Popw,Popc, Rmem,Wmem};

/*
OK - нормальное выполнение программы
Error - ошибка в программе
Empty - чтение пустого блока памяти
Все программы на ассемблере должны завершаться прерыванием
Exit
*/
enum Interrupt {OK = 0, Error=-1, Empty=2, Exit = 16, Sys =
11, Div_0=3, Lan = 32, Dev = 254};

// формат инструкции соответствует размеру блока памяти

class Block {
  public:
/* state состояние блока занят - false,
   свободен - true  */
    Block() {state = true; code = 0; nregister =0;
            operand = 0;}

    void setState(bool state) { this->state = state;}
    bool  getState() { return state;}

    void setCode(int code) { this->code = code;}
```

```
int getCode() { return code;}

void setNregister(int nregister) {
  this->nregister = nregister;}
int getNregister() { return nregister;}

void setOperand(int operand) {
  this->operand = operand;}
int  getOperand() { return operand;}

void debugBlock(){
  cout << " state="      << getState()
       << " code="       << getCode()
       << " nregister="  << getNregister()
       << " operand="    << getOperand() << endl;
}

/* формат инструкции [команда] r[номер регистра]
[$,r][номер регистра или константа]   A   B   C */
    private:
      bool state;       // блок занят или свободен
      _int8 code;       // A - байт, код команды
      _int8 nregister; // B - байт, номер регистра
      int operand;      /* C - int,  операнд [номер
                           регистра или константа] */
};

#define NUMBER_OF_REGISTERS 16
#define SIZE_OF_REGISTER_NAMES 4 /* один символ
         требуется для завершения строки strcat_s */
enum Name {r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,
r13,r14,PC,SP};
/* Определение постфиксного инкрементного operator
   для SysReg */
inline Name operator++(Name &rs, int){return
rs=(Name)(rs + 1);};

/* PC программный счетчик, указывающий на содержащий
   адрес ячейки, по которому нужно извлечь очередную
   команду */

class SysReg {
  public:
    SysReg(){
    int j = 0, i = 1; /* j регистр в массиве от 0,
                         i - номер с 1 */
    for(Name curName = r1;curName <= SP; curName++) {
```

```
      register_[j++] = new Register(i++,-1,curName);
      status = true;/* регистры не используются
                       процессом, для планирования */
    }
  }
  int getState(Name name) {
    for( int i = 0; i < NUMBER_OF_REGISTERS; i++) {
      if (register_[i]->getName() == name) {
        return (register_[i]->getState());
      }
    }
    return -1; //some system Error
  }
  void setState(_int8 nregister,int operand){
    // данные заносятся в таком виде
    for( int i = 0; i < NUMBER_OF_REGISTERS; i++) {
      if (register_[i]->getName() == nregister)  {
        register_[i]->setState(operand);
        return;
      }
    }
  }
/* очистить содержимое регистров после окончания
   программы */
  void clearSysReg() {
    for (int i = 0; i<NUMBER_OF_REGISTERS; i++){
      register_[i]->setState(-1);
    }
  }
  void Debug() { // информация о регистрах
    for( int i = 0; i < NUMBER_OF_REGISTERS; i++)
     cout << "register_["<< i <<"] "<<" numreg= "<<
          register_[i]->getNumReg() <<" state = "<<
          register_[i]->getState()  <<" name  = "<<
          register_[i]->getName()   <<" NameOf= "<<
          register_[i]->NameOf()    <<endl;
    cout <<endl;
  }

  class Register {
    public:
 /* объявление двух конструкторов: по умолчанию
    и конструктора, который устанавливает состояние и
    значение имени регистра */
    Register();
    Register(int numreg,int state, Name name){
      this->numreg=numreg;
```

```
      this->state=state;
      this->name=name;
    };

/* get and set functions.
numReg - внутреннее представление номера регистра,
например, для r1 numReg = 1, согласовано с именем */
    int   getNumReg(){return numreg;}; /* получить
                              кардинальное число */
    void  setNumReg(int numreg) {this->numreg =
    numreg;}; // установить кардинальное число
    int   getState(){return state;}; /* получить
                            состояние регистра */
    void  setState(int state){this->state=state;};
    // установить состояние регистра

    Name  getName(){return name;}; /* получить code
                                   регистра в enum */
    void  setName(Name name){this->name=name;};
    // установить code регистра в enum */

    char *NameOf(){ /* получить string представление
                    регистра */
      static char szName[SIZE_OF_REGISTER_NAMES];
      static char *Numbers[] = {
        "1", "2", "3", "4", "5", "6", "7", "8", "9",
        "10", "11", "12", "13", "14"
      }
      if ( getName() <= 13) {
         strcpy_s( szName, SIZE_OF_REGISTER_NAMES,
                   "r");
         strcat_s( szName, SIZE_OF_REGISTER_NAMES,
                   Numbers[ getName()]);
         return szName;
      }
      switch(getName()) {
        case PC:
        strcpy_s(szName,SIZE_OF_REGISTER_NAMES,"PC");
          break;
        case SP:
        strcpy_s(szName,SIZE_OF_REGISTER_NAMES,"SP");
          break;
      }
      return szName;
    }
  private:
     int numreg;
```

```
    Name name;
    int state;
  }; // end Register

  bool getStatus() {return status;}
  void setStatus(bool status) {this->status = status;}

  private:
    Register* register_[NUMBER_OF_REGISTERS];
    bool status;
    /* true - регистры не используются процессом
       false - регистры использует процесс */
}; // end SysReg


// состояния процесса
#define NUMBER_OF_STATE 7
enum State {NotRunning=0, Running=1, Blocked=2, Swapped=3,
ExitProcess=4, New=5, Ready=6};

/* New – Новый, для выполнения программы создается новый
процесс (не используется)
   Ready – Готовый (не используется)
   Running – Выполняющийся
   Blocked – Блокированный
   ExitProcess – Завершающийся
   Swapped - Выгруженный, готовый к выполнению
   NotRunning - Невыполняющийся */

class Timer {
  public:
    Timer(){time = 0;}
    static void setTime (){time = 0;}
    static int  getTime (){return time;}
    static void tick (){time++;}
  private:
  /* на операции диспетчирования затрачивается время,
  на CPU на обслуживание - ожидание и блокирование */
    static int time;
};

int Timer::time;

/* при выполнении функций класса Statistic, задается
один из параметров расчета
   TimeExec - предполагаемое время выполнения
   TimeServ - предполагаемое время обслуживания
```

```
значения StatTime присваиваются в соответствии с
индексами массива, что применяется при просмотре
вектора */
enum StatTime {TimeExecNotPreem = 0,TimeExec = 1, TimeServ =
2,  noParam = -1};
#define NUMBER_OF_PARAM 3

/* предположительное время выполнения вычисляется
методом getShortestRemainTime, который вызывается
объектом класса Dispatcher после первого наблюдения */

class VectorParam { /* вектор параметров расчета,
                       применяется в Scheduler  */
  public:
    VectorParam (){
      for( int i =0; i < NUMBER_OF_PARAM; i++){
        param[i] = new Param();
      }
    } //
    void setState(StatTime time){
      param[time]->state = true;}
    bool getState(StatTime time){
      return param[time]->state;}

    StatTime getStatTime(){
      for( int i =0; i < NUMBER_OF_PARAM; i++){
        if (param[i]->state) {
          if (i == 0) return TimeExecNotPreem;
          if (i == 1) return TimeExec;
          if (i == 2) return TimeServ;
        }
      }
      return noParam;
    } //
    void resetState(StatTime time){
         param[time]->state = false;}

    void setTthreshold(StatTime time,
                       double Tthreshold){
      param[time]->Tthreshold = Tthreshold; }
    double getTthreshold(StatTime time){
      return param[time]->Tthreshold;}

    void clearVectorParam (){
      for(int i = 0; i < NUMBER_OF_PARAM; i++) {
        param[i]->state = false;
        param[i]->Tthreshold = -1;
```

```
      }
    }
  private:
    class Param {
      public:
        Param(){Tthreshold = -1; state = false; }
        bool state; /* индикатор включения накопления
                       времени */
        double Tthreshold; /* порог предположительного
                              времени */
    };
    Param * param[NUMBER_OF_PARAM];
};

#endif
```

Модуль CPU.H

```
#ifndef __CPU_H__
#define __CPU_H__

#include "ArchitectureCPU.h"
#include "Memory.h"
#include "Device.h"
#include "LAN.h"

class CPU {
  public:
CPU(Memory &memory,int MAX_PROCESS, LAN *lan,
    Device *dev){
      this->memory = &memory;
      this->lan    = lan;
      this->dev    = dev;
    }
    ~CPU() {}

    // выполнить инструкцию - блок
    Interrupt exeInstr(int addr, SysReg *sysreg) {
      Interrupt interrupt;
      Block* block = fetch(sysreg);
      // пустой блок не декодируется
      if (block->getState()) return Error;
      // прерывание
      interrupt = decode(block, sysreg);
      switch (interrupt){
        case OK:  // изменить PC
          sysreg->setState(PC,memory->
          getNextAddr(sysreg->getState(PC)));
```

```
        break;
      case Exit: cout << "Exit" <<endl;
        return Exit;
      case Dev: // обращение к устройству печати
        dev->printData(sysreg->
          getState((Name)block->getNregister()) );
        // изменить PC
        sysreg->setState(PC, memory->
          getNextAddr(sysreg->getState(PC)));
        break;
      case Lan:  // обращение к устройству сети
        lan->sendData(sysreg->
          getState((Name)block->getNregister()) );
        // изменить PC
        sysreg->setState(PC, memory->
        getNextAddr(sysreg->getState(PC)));
        break;
      default : return interrupt;
    }
    return OK;
  } // end exeInstr

  void Debug(SysReg *sysreg){ /* трассировка
         состояния регистров для одного блока
          см. Scheduler Debug */
    sysreg->Debug();
    // только один блок
    Block* block = fetch(sysreg);
    block->debugBlock();
    decode(block, sysreg);
    sysreg->Debug();
  }
private:
  Memory *memory;
  int MAX_PROCESS;

  // функции алгоритма CPU
  // выборка инструкции по состоянию PC
  Block* fetch(SysReg *sysreg) {
    return memory->read(sysreg->getState(PC));
  }

  LAN *lan;
  Device *dev;
protected:
  virtual Interrupt decode(Block* block,
                           SysReg *sysreg) {
```

```
/* доступ к выборке операнда access:
   sysreg[id]->getState(....)  */
      Block* mblock = NULL; /* используется при
                               чтении и записи */
      switch( block->getCode()){
        case Mov: /* Перемещение содержимого второго
                     операнда в первый {регистр} */
          sysreg->setState(block->getNregister(),
                           block->getOperand());
          break;
        case Add :
 /* 1. взять содержимое регистра и операнда (тоже регистр)
    2. результат записать в первый операнд */
       sysreg->setState((Name)block->getNregister(),
       sysreg->getState((Name)block->getNregister())+
                              block->getOperand()  );
          break;
        case Sub :
/* 1. взять содержимое регистра и операнда (тоже регистр)
   2. результат записать в первый операнд */
       sysreg->setState( (Name)block->getNregister(),
       sysreg->getState((Name)block->getNregister())-
                              block->getOperand()  );
          break;
        case Mul :
/* 1. взять содержимое регистра и операнда (тоже регистр)
   2. результат записать в первый операнд */
       sysreg->setState( (Name)block->getNregister(),
       sysreg->getState((Name)block->getNregister())*
       sysreg->getState((Name)block->getOperand()) );
          break;
        case Div :
          if (sysreg->getState((Name)block->
               getOperand()) == 0) return Div_0;
/* 1. взять содержимое регистра и операнда (тоже регистр)
   2. результат записать в первый операнд */
     sysreg->setState( (Name)block->getNregister(),
     sysreg->getState((Name)block->getNregister()) /
     sysreg->getState((Name)block->getOperand() ) );
          break;
        case Int : // прерывание
          return (Interrupt) block->getOperand();
          break;
        case Wmem:
        // Сохранить содержимое регистра в память
          mblock = memory->read(block->getOperand());
          if (!mblock ->getState())return Error;
```

```
            // чтение занятого блока
            mblock ->setOperand( sysreg->
            getState((Name)block-> getNregister()));
            mblock ->setState(false); // занят
            break;
          case Rmem:
           // чтение операнда из блока памяти в регистр
            mblock = memory->read(block->getOperand());
            if (mblock ->getState())return Empty;
            // чтение только занятого блока
            sysreg->setState((Name)block->
            getNregister(), mblock->getOperand());
            break;
          default : return Error;
      }
      return OK;
    } // end decode
};
#endif
```

Модуль Memory.h

```
#ifndef __Memory_H__
#define __Memory_H__

#include <iostream>
using namespace std;
#include "ArchitectureCPU.h"

class Memory {
  public:
    Memory(int SIZE_OF_MEMORY_IN_BLOCKS){
      currBlock = 0;
      this->SIZE_OF_MEMORY_IN_BLOCKS =
                          SIZE_OF_MEMORY_IN_BLOCKS;
      heap = new Block[SIZE_OF_MEMORY_IN_BLOCKS];
    }
  /* применяется при считывании инструкций из файла
     функция устанавливает начальный адрес для
     последовательной записи  */
    void setAddrBlock( int addr) {currBlock = addr; }

  // применяется при считывании инструкций из файла
    void setCmd(int cmd, int op1, int op2) {
        int addr = currBlock++;
        if (addr == -1) {
            cout << "No free blocks" << endl;
```

```
            return;
        }
          heap[addr].setState(false);
          heap[addr].setCode(cmd);
          heap[addr].setNregister(op1);
          heap[addr].setOperand(op2);
    }

  /* read применяется при считывании инструкций CPU и при
  организации свопинга с виртуальной памятью */
    Block* read(int addr) { return &heap[addr]; }
    /* получить индекс следующего блока,
       если нет блока, то Error  */

    // использовать функции  для управления памятью
    int getAddrFreeBlock(){ /* выбрать первый свободный
                               блок при просмотре */
      for (int i = 0; i < SIZE_OF_MEMORY_IN_BLOCKS; i++)

        if (heap[i].getState()) return i;
      return -1;  // нет свободных блоков
    }

    /* адрес вычисляется как следуюший по индексу;
  для страничной и сегментной организации метод должен быть
  замещен. Используется в CPU, для содержимого регистра PC */
    int getNextAddr(int currAddr){return currAddr+1;}

    void clearMemory() {  // очистка всей памяти
      for (int i = 0; i < SIZE_OF_MEMORY_IN_BLOCKS; i++)
         clearBlock(i);
      currBlock = 0;
    }

   void clearMemory(int AddrFrom, int AddrTo) {
   // очистка части памяти, текущий блок не меняется
   for(int i=AddrFrom;i<SIZE_OF_MEMORY_IN_BLOCKS;i++) {
     if (i > AddrTo) return;
     clearBlock(i);
   }
   }

   int getSizeMemory(){
      return SIZE_OF_MEMORY_IN_BLOCKS;}

  // можно посмотреть содержимое памяти
  void debugHeap() {
```

```
    for (int i = 0;i < SIZE_OF_MEMORY_IN_BLOCKS; i++){
    cout << "heap[" << i <<"] = " <<" state="<<
    heap[i].getState()            << " code="<<
    heap[i].getCode()             << " nregister=" <<
    heap[i].getNregister()        << " operand="   <<
    heap[i].getOperand() << endl;
    }
    }
  private:
    void clearBlock(int addr) {
      heap[addr].setState(true);
      heap[addr].setCode(0);
      heap[addr].setNregister(0);
      heap[addr].setOperand(0);
    }
    int SIZE_OF_MEMORY_IN_BLOCKS;
    int currBlock;
    Block *heap;
};

class Code: public Memory {
  public:
    Code(int
    SIZE_OF_PROGRAMM):Memory(SIZE_OF_PROGRAMM){}

    virtual ~Code(){cout<<" object Code deleted"<< endl;}
};

#endif
```

Модуль Device.h

```
#include <iostream>
using namespace std;

class Device {
  public:
    Device(){}
    void printData(int data){
      cout << "Device printData: data = " << data << endl;
    }
  private:
  protected:
};
```

Модуль LAN.h

```
#include <iostream>
using namespace std;
```

```
class LAN {
  public:
    LAN(){}
    void sendData (int data){
      cout << "LAN send: data = " << data << endl;
    }
  private:
  protected:
};
```

Модуль HANDLE.h

```
#ifndef __HANDLE_H__
#define __HANDLE_H__

class HANDLE {
  public:
    HANDLE(){
      ID = -1; /* идентификатор присваивается
      системой, -1 - идентификатор не присвоен */
      // process scheduling
      Tenter = -1;  // Время входа
      Tbegin = -1;  // Время начала
      Tservice = 0; // Время обслуживания в очередях
      Texec    = 0; // Время выполнения
      Tterminate = 0; // Время завершения
    }
    virtual ~HANDLE(){
    cout<< " object HANDLE deleted" << endl;}

    int getTenter(){return Tenter;}
    int getTbegin(){return Tbegin;}
    int getTservice(){return Tservice;}
    int getTterminate(){return Tterminate;}
    int getTexec(){return Texec;}
    int getTround(){return Texec + Tservice;} /* Время
                                                 оборота */
    float getTnorm(){return (float)(Texec + Tservice)
/ Tservice;}

    void ProcessTime(){
cout <<"      ProcessTime ID = "<< ID << endl;
cout <<"Tenter:    "<< Tenter <<endl; // Время входа
cout <<"Tbegin:    "<< Tbegin <<endl; // Время начала
// Время обслуживания в очередях
cout <<"Tservice:  "<< Tservice <<endl;
// Время завершения
cout <<"Tterminate:"<< Tterminate<<endl;
cout <<"Texec:     "<<Texec<<endl; // Время завершения
```

```
// Время оборота
cout <<"Tround:    "<<getTround()<<endl;
// Нормализованное время
cout <<"Tnorm:     "<<getTnorm()<<endl;
   }

  private:
    int ID;
    // process scheduling
    int Tenter;      // Время входа
    int Tbegin;      // Время начала
    int Tservice;    // Время обслуживания в очередях
    int Tterminate; // Время завершения
    int Texec;       // Время выполнения
  protected:
    int getID(){return ID;}
    void setID(int ID){this ->ID = ID;}
    // process scheduling
    void setTenter(int Tenter){this ->
      Tenter = Tenter;}
    void setTbegin(int Tbegin){this ->
      Tbegin = Tbegin;}
    void setTservice(int Tservice){this ->
      Tservice = Tservice;}
    void setTterminate(int Tterminate){this->
      Tterminate = Tterminate;}
    void setTexec(int Texec){this ->Texec = Texec;}
    void clearTime(){ // применяется при завершении
      Tenter = -1;     // Время входа
      Tbegin = -1;     // Время начала
      Tservice = 0;    // Время обслуживания
      Texec = 0;       // Время выполнения
      Tterminate = 0; // Время завершения
    }
};

// для CriticalSection - критической секции
class CS: HANDLE {
  public:
    CS(){ cs = false;
      // false - критическая секция не занята
      // true  - критическая секция  занята
    }
    virtual ~CS(){cout<<" object CS deleted"<< endl;}
    bool getCS(){return cs;}
    void setCS(bool cs){this->cs = cs;}
  private:
```

```
    bool cs;
  protected:
};
#endif
```

Модуль Process.h

```
#ifndef __Process_H__
#define __Process_H__

#include "HANDLE.h"
#include "ArchitectureCPU.h"

/* состояния процесса см.
State {New, Ready, Running, Blocked, ExitProcess,
NotRunning};
New – Новый, для выполнения программы создается новый процесс
Ready – Готовый
Running – Выполняющийся
Blocked – Блокированный
ExitProcess – Завершающийся
NotRunning - Невыполняющийся */

// реализация модели из двух состояний
class Process: public HANDLE {
  public:
    Process(){
      this->user = "";
      state = NotRunning;
    }
    virtual ~Process(){
        cout<< " object Process deleted" << endl;}

    string getUser(){return user;}
    void setUser(string user){this->user = user;}
    int getID(){return HANDLE::getID();}

    void setID(int ID){HANDLE::setID(ID);}

    // состояние процесса
    void setState(State state){this->state=state;}
    State getState(){ return state;}
  private:
    State state;  // модель состояния процесса
  protected:
    string user;
};

class PCB: public Process { // Process Control Block
  public:
```

```
PCB(){
  sysReg = NULL;
  addr = -1; /* адрес размещения в оперативной
                памяти начала программы */
  virtualAddr = -1; /* адрес размещения в
                       виртуальной памяти */
  // Scheduling
  timeSlice = -1; /* квота времени, выделенная
                     процессу для метода RR */
  priority  = -1; /* приоритет процесса может
                     быть изменен системой DP */
}
virtual ~PCB(){
cout<<" object PCB deleted" << endl;}

void setSysReg(SysReg *sysReg){
           this->sysReg=sysReg;}
SysReg * getSysReg(){ return sysReg;}

// адрес размещения в оперативной памяти
int getAddr(){return addr;}
void setAddr(int addr){this ->addr = addr;}

// адрес размещения в виртуальной памяти
int getVirtualAddr(){return virtualAddr;}
void setVirtualAddr(int virtualAddr){
     this ->virtualAddr = virtualAddr;}

// приоритет процесса
int getPriority(){return priority;}
void setPriority(int priority){
           this ->priority = priority;}

// квант непрерывного времени для процесса
int getTimeSlice(){return  timeSlice;}
void setTimeSlice(int  timeSlice){
       this -> timeSlice =  timeSlice;}

// получить string представление состояния
char *NameOf(State state){
  const int SIZE_OF_STATE_NAME = 15;
  static char szName[SIZE_OF_STATE_NAME];
  switch(state) {
    case NotRunning:
strcpy_s( szName, SIZE_OF_STATE_NAME,"NotRunning");
      break;
```

```
        case Running:
   strcpy_s( szName, SIZE_OF_STATE_NAME, "Running");
          break;
        case Swapped:
   strcpy_s( szName, SIZE_OF_STATE_NAME, "Swapped" );
          break;
        case New:
   strcpy_s( szName, SIZE_OF_STATE_NAME, "New" );
          break;
        case Ready:
   strcpy_s( szName, SIZE_OF_STATE_NAME, "Ready" );
          break;
        case Blocked:
   strcpy_s( szName, SIZE_OF_STATE_NAME, "Blocked" );
          break;
        case ExitProcess:
   strcpy_s(szName, SIZE_OF_STATE_NAME,"ExitProcess");
          break;
      }
      return szName;
    };
  private:
    int addr;
    int virtualAddr;
    SysReg *sysReg;
    int timeSlice; /* квота времени, выделенная
                      процессу для метода RR */
    int priority;  // приоритет процесса
  protected:
};

class ProcessImage: public PCB  {
  public:
    ProcessImage(){
      status = true; /* ProcessImage в
                        VirtualMemory свободен */
      this->memory=NULL;
      flag = 1; // ресурс свободен
    }
    virtual ~ProcessImage(){
      cout<< " object ProcessImage deleted" << endl; }
    void Debug () {
      cout <<"-- begin Debug ProcessImage --" << endl;
      cout << "user: "<< user << endl;
      cout << "ID:   "<< getID()  << endl;
      cout << "addr: "<< getAddr() << endl;
```

```
    cout << "VirtualAddr:"<<getVirtualAddr()<< endl;
    cout << "State: "<< NameOf(getState())  << endl;
    cout << "priority: "<< getPriority()<< endl;
    cout << "timeSlice:"<< getTimeSlice()<< endl;
    cout << "status:   "<< status<< endl;
    cout << "flag:     "<< flag<< endl;
    cout << "SysReg:   "<< endl;
      SysReg * sysReg = getSysReg();
      if ( sysReg == NULL )
        cout << "SysReg: NULL"<< endl;
      else {
        sysReg->Debug();
      }
    cout << "Code:   "<< endl;
      if ( memory==NULL )
        cout << "Code (memory): NULL"<< endl;
      else memory->debugHeap();
    cout << "--- end Debug ProcessImage --"<<endl;
  }

  void DebugTime () {
    cout<<" -- ProcessImage: DebugTime ---"<< endl;
    cout<<" ID = "  <<  getID()<< endl;
    cout<<" Timer tick = " <<Timer::getTime()<< endl;
    cout<<" TimeSlice  = " << getTimeSlice()<< endl;
    cout<<" Texe       = " << getTexec()<< endl;
    cout<<" Priority   = " << getPriority()<< endl;
  }

  Memory* getCode(){return memory;}
  void setCode(Memory* memory){
                            this ->memory = memory;}

/* статус образа процесса, используется только в
   классе VirtualMemory для выделения памяти: true
   ProcessImage свободен */

  void setStatus(bool status){this->status=status;}
  bool getStatus(){ return status;}

  /* флаг, для моделирования доступа и ожидания
     (блокирования) ресурса, требуемому процессу для
     выполнения */
  void setFlag(int flag){this->flag=flag;}
  int getFlag(){ return flag;}

  void clearTime(){HANDLE::clearTime();}
```

```
  private:

    Memory* memory;
    // для выделения памяти VirtualMemory
    bool status;
    int flag; /* доступ к ресурсу, например 1 - ресурс
                 свободен, 0 – занят */
  protected:
    friend class Dispatcher;
    friend class Scheduler;
};
#endif
```

Модуль Dispatcher.h

```
#ifndef __Dispatcher_H__
#define __Dispatcher_H__

#include <iostream>
using namespace std;
#include "ArchitectureCPU.h"
#include "VirtualMemory.h"
#include "Scheduler.h"
#include "Process.h"

class Dispatcher { /* управляет ресурсами, изменяет
                      состояние процессов */
  public:
    Dispatcher (int SIZE_OF_VIRTUAL_MEMORY_IN_IMAGES,
                Scheduler * scheduler,MMU * mmu){
      virtualMemory = new VirtualMemory(
                SIZE_OF_VIRTUAL_MEMORY_IN_IMAGES);
      this -> scheduler = scheduler;
      this -> mmu = mmu;       // Model memory unit
      this -> cs  = new CS;    // critical section
      this -> timeSlice = -1; /* квант времени для
                          планирования по методу RR */
    }

    ProcessImage * getVirtualMemory(){ /* системные
      регистры не присвоены, процессор не выделен */
      // 1. выделить virtualMemory
      int addr = virtualMemory->getAddrFreeImage();
      if (addr == -1) return NULL;
      virtualMemory->setAddrImage(addr);
      // 2. получить образ по виртуальному адресу
      ProcessImage * processImage =
                          virtualMemory->getImage();
```

```
  // 3. сохранить виртуальный адрес в образе
  processImage->setVirtualAddr(addr);
  // 4. увеличить время обслуживания
  processImage->
       setTservice(processImage->getTservice()+1);
  return processImage; /* по установленному
                          адресу */
}

void freeVirtualMemory(HANDLE *handle){
  ProcessImage * processImage =
                           (ProcessImage *)handle;
  /* 0. только если процесс в очереди
     ExitProcess (выполнился) */
  State state = processImage->getState();
  int ID = processImage->getID();
  // 1. очистить регистры
  scheduler->getSysReg(ID)->setStatus(true);
  scheduler->getSysReg(ID)->clearSysReg();
  processImage->setSysReg(NULL);
  // 2. удалить из очереди
  scheduler->pop(state, processImage);
  // 3. сохранить наблюдение для метода SPN;
  if (scheduler->vParam.getStatTime() !=
   noParam )
     {scheduler->setObservation(processImage);
   if (scheduler->
            vParam.getState(TimeExecNotPreem))
            // выполнения SPN
       scheduler->setTthreshold(TimeExec);
    /* 4. установить порог наименьшего
          ожидаемого времени */
    if (scheduler->vParam.getState(TimeExec))
    // выполнения SRT
      scheduler->setTthreshold(TimeExec);
    if (scheduler->vParam.getState(TimeServ))
      scheduler->setTthreshold(TimeServ);
      // обслуживания HRRN
   }
   /* 4. освободить virtualMemory,
         инициализировать PCB */
   virtualMemory->
     clearImage(((PCB*)handle)->getVirtualAddr());
}

void initProcessID(ProcessImage *processImage,
                   Memory *code){
```

```
     // 1. определить ID
      int ID = scheduler->getID();
      if (ID >= 0){ //ID не -1, т.е. массив регистров
        processImage->setID(ID);
      } else { // в очередь как новый
    cout<<"init Process ID > MAX_PROCESS in system"<<
    endl;
      }
     // 2. выделить регистры по ID
    scheduler->getSysReg(ID)->setStatus(false);
    processImage->setSysReg(scheduler->getSysReg(ID));
     // 3. состояние процесса
     processImage->setState(NotRunning);
     // 4. разместить программу
     processImage->setCode(code);
     code = NULL; // к программе нет доступа из вне
     // 5. установить время входа
     processImage->setTenter(Timer::getTime());
     // 6. инициализировать время обслуживания
     processImage->setTservice(0);
     // 7. в очередь как NotRunning
     scheduler->push(NotRunning, processImage);
    }

    /* номера пунктов приведены в соответствии с
       порядком изучения */
    virtual void dispatch(){
      ProcessImage * processImage = NULL;
      Timer::tick(); // увеличить время
/* 1. просмотреть всю очередь Running, если
есть завершившиеся процессы, убрать только из
очереди в виртуальной памяти; объект образа
удаляется только методом ядра */
      int size_ = scheduler->size(Running);
      for (int i = 0; i < size_; i++){
        processImage = (ProcessImage *)(scheduler->
                                  front(Running));
        // увеличить время обслуживания
        processImage->setTservice(processImage->
                                 getTservice()+1);
        if (processImage->getState() ==
                 ExitProcess) {  // процесс завершен
        scheduler->pop(Running);// удаляем из очереди
        scheduler->push(ExitProcess, processImage);
        // ставим в очередь выполненных
        }
      }
```

```
// 3. просмотреть очередь Blocked
      size_ = scheduler->size(Blocked);
      for (int i = 0; i < size_; i++){
        processImage = (ProcessImage *)(scheduler->
                                        front(Blocked));
        // увеличить время обслуживания
        processImage->setTservice(processImage->
                                      getTservice()+1);
       if(!cs->getCS()){// критическая секция свободна
        scheduler->pop(Blocked);// удаляем из очереди
        processImage->setState(Running); /* изменяем
                                  состояние процесса */
        processImage->setFlag(1);
        // не изменяем приоритет процесса
        scheduler->push(Running,
         processImage);// ставим в очередь выполняемых
        cs->setCS(true); // закрыть секцию
        break;
       }
      }
/* 2. просмотреть очередь NotRunning и, если для
образа определены все ресурсы (регистры и т.д.)
поставить на выполнение */
      size_ = scheduler->size(NotRunning);
      for (int i = 0; i < size_; i++){
        processImage =  (ProcessImage *)(scheduler->
                                  front(NotRunning));
        // увеличить время обслуживания
        processImage->setTservice(processImage->
                                    getTservice()+1);
        /* выделены ли под образ регистры системы и
           процессор */
        if (processImage->getSysReg() == NULL) {
           /* выделить регистры, если нет свободных,
              оставить в очереди */
          SysReg *sysReg = scheduler->
                      getSysReg(scheduler->getID());
          if (sysReg == NULL) {
            scheduler->pop(NotRunning); /* удаляем из
                                           очереди */
            scheduler->push(NotRunning, processImage);
            // ставим в эту же очередь
            continue;
          }
          processImage->setSysReg(sysReg);
        }
```

```
    scheduler->pop(NotRunning); // удаляем из очереди
/* 2.1. Code загрузить в память, процессор выделен
обратиться к блоку управления памятью, если есть ресурс */
        if (processImage->getFlag() == 1) {
          processImage->setState(Running);/* изменяем
          состояние процесса; установить квант времени и
          приоритет, в зависимости от метода */
          processImage->setTimeSlice(this->timeSlice);
// метод RR
          if ( prSlice.getPrioritySlice()){
// метод DP
            int pr = processImage->getPriority();
            processImage->
               setTimeSlice(prSlice.getTimeSlice(pr));
            processImage->
               setPriority(prSlice.getPriority(pr));
          }
          scheduler->push(Running, processImage);
          // ставим в очередь выполняемых
        } else {
        processImage->setState(Blocked); /* изменяем
                                  состояние процесса */
        scheduler->push(Blocked, processImage);
        // ставим в очередь блокированных
        }
      } // end for
/* 3. просмотреть очередь Swapped и применить метод DP
      выполняется только для метода DP
      bool priority = prSlice.getPrioritySlice();
      if ( priority ){ // метод DP
        size_ = scheduler->size(Swapped);
        for (int i = 0; i < size_; i++){
          processImage = (ProcessImage *)(scheduler->
                                      front(Swapped));
// 3.1. изменить приоритет процесса и TimeSlice
          int pr = processImage->getPriority();
          processImage->
               setTimeSlice(prSlice.getTimeSlice(pr));
          processImage->
               setPriority( prSlice.getPriority(pr));
          scheduler->pop(Swapped); /* удаляем из
                                      очереди */
          scheduler->push(Swapped, processImage);
          // ставим в эту же очередь
        }
      }
```

```
      // подготовить работу
      if ( scheduler->empty(Running) && scheduler->
                                    empty(Swapped) ) {
        cout << "dispatch: empty Running && Swapped"
        <<endl;
        return;//нет заданий на выполнение
      }
      scheduleProcess(mmu, priority);

/* выполнить процессы в соответствии со
сформированным заданием */
      Interrupt interrupt = executeProcess(mmu);
    }

/* выполнить процессы в соответствии со
сформированным заданием */
    virtual void scheduleProcess(MMU * mmu,
                                 bool priority){
        bool maxProcess = scheduler->
                          scheduleJob(mmu,priority);
/* задание сформировано
   maxProcess  == true превышение MAX_PROCES,
   есть еще процессы
   maxProcess  == false, все процессы вошли в задание */
    }

/* выполнить процессы в соответствии со
сформированным заданием */
    virtual Interrupt executeProcess (MMU * mmu)
              {return scheduler->execute(mmu);}

    // для управления критической секцией
    HANDLE * getHandleCS(){ return (HANDLE *)cs; }

    void EnterCriticalSection(HANDLE* handle){
      if (!cs->getCS()) { // ресурс не занят
    // для этого процесса критическая секция открыта
        ((ProcessImage *) handle)->setFlag(1);
        cs->setCS(true); /* закрыть критическую
                            секцию для других */
      }
      else {
    // закрыть критическую секцию для этого процесса
        ((ProcessImage *) handle)->setFlag(0);
      }
    }
```

```
    void LeaveCriticalSection(HANDLE* handle){
      cs->setCS(false); /* открыть критическую секцию,
                           не занят, false */
    }

    // для планирования по методу RR
    void setTimeSlice(int timeSlice){this ->
                                timeSlice = timeSlice;}
    void  resetTimeSlice(){
      this ->timeSlice = -1;
      this->prSlice.setPrioritySlice(-1,NULL);
    }

/* для планирования по методу DP – динамических
                                        приоритетов */
    void setTimeSlice(int size, int * prSlice){
      this->prSlice.setPrioritySlice(size,prSlice);
    }

// для метода SRT и HRRN
    void setTpredict(StatTime time){
      scheduler -> vParam.setState(time);}
    void resetTpredict() {
      scheduler -> vParam.clearVectorParam();}
    double getTpredict(string user,StatTime time) {
      return scheduler->getTpredict(user,time);}

    void DebugQueue(State state){
      scheduler-> DebugQueue(state);}
    void DebugVirtualMemory(){
      this->virtualMemory -> Debug();}
    void DebugPCB(HANDLE* handle){
      handle -> ProcessTime();
      ((ProcessImage *)handle)->Debug(); }

    class PrioritySlice {
      public:
        PrioritySlice(){
          this-> size = -1;
          this-> prSlice = NULL;
        }
        void setPrioritySlice(int size,
                                int * prSlice){
          this-> size = size;
          this-> prSlice = prSlice;
        }
        bool getPrioritySlice () {
          if (prSlice == NULL) return false;
```

```
        return true;
      }

      int getTimeSlice(int priority) {
        if (size == -1) return -1; /* timeSlice не
                                      установлен */
        if (priority == -1)  return prSlice[0];
        // оставляем последний приоритет
        if (priority == size-1)
          return prSlice[size-1];
        return prSlice[priority+1];
      }
      int getPriority(int priority) {
        if (priority == -1)  return 0;
        if (priority == size-1) return size-1;
        return priority+1;
      }

    private:
      int size;
      int * prSlice; /* массив приоритетов и
  квантов времени для планирования по методу DP */
  };

protected:
  VirtualMemory * virtualMemory;
  Scheduler* scheduler;
  MMU* mmu;
  CS* cs;         // critical section
  int timeSlice; /* квант времени для планирования
                    по методу RR */
  PrioritySlice prSlice; /* для планирования по
                            методу DP */
};
#endif
```

Модуль Kernel.h

```
#ifndef __Kernel_H__
#define __Kernel_H__

#include "Dispatcher.h"
#include "Scheduler.h"

class Kernel {
  public:
    Kernel(Dispatcher *dispatcher){
      this ->dispatcher = dispatcher;
    }
```

```
    // создание процесса
    HANDLE * CreateProcess(string user, Memory *code){
      /* обратиться к блоку управления VirtualMemory
         для выделнения памяти под образ процесса */
      ProcessImage *processImage = dispatcher ->
                                   getVirtualMemory();
      if (processImage==NULL) return NULL; // нет памяти
      processImage->setUser(user);
      dispatcher->initProcessID(processImage,code);
      // инициализировать процесс
      return (HANDLE *)processImage;
    } // end CreateProcess

    // удаление процесса
    void TerminateProcess(HANDLE* handle){
    // 1. очистить VirtualMemory и sysreg процесса
      dispatcher -> freeVirtualMemory(handle);
    }
    // методы критической секции
    HANDLE* CreateCriticalSection(){
      return dispatcher->getHandleCS(); }

    void EnterCriticalSection(HANDLE* handle){
      dispatcher -> EnterCriticalSection(handle);
    }
    void LeaveCriticalSection(HANDLE* handle){
      dispatcher -> LeaveCriticalSection(handle);
    }

    void DebugProcessImage(HANDLE *handle){
      ((ProcessImage *)handle)->Debug();}
  private:
    Dispatcher *dispatcher;
  protected:
};

#endif
```

Модуль MMU.H

```
#include "Memory.h"
#include "Process.h"

class MMU {
  public:
    MMU(Memory &memory) {this ->memory = &memory;}
    /* загрузка программы из образа процесса в
       основную память */
    virtual void swapIn(ProcessImage * processImage){
```

```
  // получить программу из образа процесса
  Memory * code = processImage->getCode();
  /* получить адрес размещения программы в
     оперативной памяти, если он был */
  int addrPCB  = processImage->getAddr();
  int addrReal = getRealAddr(); /* адрес
                          размещения программы */
  if (addrPCB == -1)  // не присвоен
    /* если PC = -1, то PC не проинициализирован
       присвоить начало программы
    processImage->getSysReg()->
                          setState(PC,addrReal);
  else {
  /* размещение програмы в памяти
     1. определить смещение PC - addrPCB, изменить
        адрес PC */
   int addrOffPC = processImage->getSysReg()->
                          getState(PC) - addrPCB;
   processImage->getSysReg()->
              setState(PC,addrOffPC + addrReal);
  }
  // запомнить адрес в блоке PCB
  processImage->setAddr(addrReal);
  /* установить адрес размещения программы в
     оперативной памяти */
  setAlloc(addrReal);
// прочитать блок кода, начало программы в образе
  Block * block = NULL;
  for(int i=0; i < code->getSizeMemory(); i++){
    block = code->read(i); /* код программы
                          читается с 0 адреса */
    /* записать блок в память только
       последовательно, на незанятые блоки */
    if (! block->getState()) {
      memory->setCmd(block->getCode(),
                     block->getNregister(),
                     block->getOperand() );
    }
  }
}

/* выгрузка программы из основной памяти,
   применяется в алгоритмах вытеснения */
virtual void swapOut(
               ProcessImage * processImage){
/* получить адрес размещения программы в
   оперативной памяти из PCB */
```

```
      int AddrFrom = processImage->getAddr();
      int AddrTo = AddrFrom + processImage->
                        getCode()-> getSizeMemory() -1;
      memory -> clearMemory(AddrFrom,AddrTo);
    }

    /* определить свободный адрес для загрузки
       программы ограничения, проверяется только один
       блок, остальные считаются свободными - это не
       всегда так */
    virtual int getRealAddr(){return this->memory->
                                  getAddrFreeBlock();}
    // установить адрес размещения программы в памяти
    virtual void setAlloc(int addr){memory->
                                  setAddrBlock(addr);}

    // показать выполнение swapIn
    virtual void Debug(HANDLE * handle){
      // получить программу из образа процесса
      ProcessImage * processImage =
                                (ProcessImage *)handle;
      // см. void swapIn(ProcessImage * processImage)
      Memory * code = processImage->getCode();
      /* получить адрес размещения программы в
         оперативной памяти, если он был */
      cout << endl;
      cout <<
"-get address of loading programm in memory & set PC"
      <<endl;
      int addrPCB = processImage->getAddr();
      cout << "addrPCB = " << addrPCB <<
      " address of loading programm from PCB" <<endl;
      int addrReal = getRealAddr(); /* адрес
                                размещения программы */
      cout << "addrReal= " << addrReal <<
      " address of loading programm Real"<<endl;

      if (addrPCB == -1) /* не присвоен, если PC = -1,
        то PC не проинициализирован, присвоить начало
        программы */
        cout <<
        " if address in PCB == -1 set addrReal PC = "
         << addrReal<< endl;
      else {  /* размещение програмы в памяти
          1. определить смещение  PC - addrPCB и
             изменить адрес PC */
        cout <<"PC = "<<processImage->getSysReg()->
```

```
            getState(PC)<<" address in PC "<<endl;
        int addrOffPC = processImage->getSysReg()->
                              getState(PC) - addrPCB;
        cout <<"addrOffPC = PC - addrPCB = "<<
                   addrOffPC<<" address offset"<<endl;
        cout <<"PC = addrOffPC + addrReal = "<<
                         addrOffPC + addrReal <<endl;
      }
     // прочитать блок кода, начало программы в образе
      Block * block = NULL;
      cout << endl;
      cout <<
      "read block from process image code to memory"
      <<endl;
      cout << " size code = "<< code->
                              getSizeMemory()<<endl;
      for(int i=0; i < code->getSizeMemory(); i++){
        block = code->read(i); /* код программы
        читается с 0 адреса, записать  блок в память
        только последовательно, на незанятые блоки */
        if (! block->getState()) {
          block->debugBlock();
        }
      }
    }

    virtual void DebugMemory(){memory->debugHeap();}
  private:
    Memory *memory;
};
```

Модуль VirtualMemory.h

```
#ifndef __VirtualMemory_H__
#define __VirtualMemory_H__

#include <iostream>
using namespace std;
#include "ArchitectureCPU.h"
#include "Process.h"

class VirtualMemory {
  public:
    VirtualMemory(int SIZE_OF_VIRTUAL_MEMORY_IN_IMAGES){
    this->SIZE_OF_VIRTUAL_MEMORY_IN_IMAGES =
                  SIZE_OF_VIRTUAL_MEMORY_IN_IMAGES;
    image =
      new ProcessImage[SIZE_OF_VIRTUAL_MEMORY_IN_IMAGES];
```

```
    currImage = 0;
    }

 /* применяется при считывании инструкций
    функция устанавливает начальный адрес для
    последовательной записи */
  virtual void setAddrImage(int addr){currImage = addr;}

  // получить по установленному адресу образ процесса
  ProcessImage * getImage() {return &image[currImage];}

   /* применяется при инициализации образа
      по установленному адресу, записать в виртуальную
      память программу  */
  virtual void setMemory(Memory * memory) {
    image[currImage].setStatus(false); // образ занят
    image[currImage].setCode(memory);
  }

   /* read применяется при организации свопинга:
      из виртуальной памяти в оперативную */
  virtual Memory* read(int addr) {
    return image[addr].getCode();}

   /* получить индекс следующего образа, если нет
      образа, то Error, -1 использовать функции для
      управления памятью */
  virtual int getAddrFreeImage(){/* выбирается первый
                       свободный образ при просмотре */
  for (int i= 0;i<SIZE_OF_VIRTUAL_MEMORY_IN_IMAGES;i++)
    if (image[i].getStatus()) {
      // true VirtualMemory  образ свободен
      image[i].setStatus(false);
      return i;  // false занят
    }
    return -1; // нет свободных образов
  }

   /* при удалении программы из системы:
      освободить virtualMemory, инициализировать PCB */
   virtual void clearImage(int addr) {
     image[addr].setStatus(true);
     image[addr].setCode(NULL);

     image[addr].setAddr(-1);
     image[addr].setID(-1);
     image[addr].setPriority(-1);
```

```
      image[addr].setTimeSlice(-1);
      image[addr].setState(NotRunning);
      image[addr].setSysReg(NULL);
      image[addr].setUser("");
      image[addr].setVirtualAddr(-1);
      image[addr].clearTime();
      image[addr].setTimeSlice(-1);
     }

   virtual void clearMemory() {
   for(int i=0; i<SIZE_OF_VIRTUAL_MEMORY_IN_IMAGES;i++)
     clearImage(i);
   currImage = 0;
   }

   /* можно посмотреть содержимое образа процесса в
      виртуальной памяти */
   virtual void Debug() {
   for(int i=0;i<SIZE_OF_VIRTUAL_MEMORY_IN_IMAGES;i++){
    cout<<" VertualAddress = "<<i<<" status = "<<
    image[i].getStatus()<<" ID = "<< image[i].getID()
     << endl;
    }
    }
  private:
    int SIZE_OF_VIRTUAL_MEMORY_IN_IMAGES;
    int currImage;
    ProcessImage *image;
};

#endif
```

Модуль Scheduler.h

```
#ifndef __Scheduler_H__
#define __Scheduler_H__

#include "ArchitectureCPU.h"
#include "Process.h"
#include "Statistic.h"
#include <queue>
#include <hash_map>
#include <valarray>

using namespace stdext; // for hash_map
using namespace System; // for Math::

/* задание для выполнения процессором CPU программ
   в многозадачном режиме */
```

```
class Job {
  public:
    Job (){ // конструктор по умолчанию
      this->addr = -1;
      this->id = -1;
      this->done = false; // выполнен
      this->processImage = NULL; /* указатель на
                                    образ процесса */
    }
    void set(int addr, int id, bool done){
      this->addr = addr;
      this->id = id;
      this->done = done;
    }
    void set(int addr, int id, bool done,
                  ProcessImage * processImage){
      this->addr = addr;
      this->id = id;
      this->done = done;
      this->processImage = processImage;
    }

    void setAddr(int addr){this->addr = addr;}
    int  getAddr(){return addr;}

    void setId  (int id  ){this->id   = id;}
    int  getId  (){return id;}

    void setDone(bool done){this->done= done;}
    bool getDone(){return done;}

    void setProcessImage(ProcessImage * processImage)
    { this->processImage = processImage;}
    ProcessImage * getProcessImage(){
      return processImage;}

    void Debug() {
        cout << " job: addr = "  << addr
                  << " id   = "  << id
                  << " done = "  << done << endl;
    }
  private:
    int addr;// начальный адресс выполнения программы
    int id;  // идентификатор процесса
    bool done;  // индикатор выполнения задания
    ProcessImage * processImage; // образ процесса
};
```

```
class Scheduler {
  public:
    Scheduler(CPU *cpu, int MAX_PROCESS){
      this ->cpu = cpu;
      /* каждый процесс виртуально владеет своими
         регистрами */
      this->MAX_PROCESS = MAX_PROCESS;
      /* в многозадачном режиме, после прерывания для
         каждого процесса сохраняется состояние
         регистров, т.е. каждый процесс виртуально
         владеет своими регистрами */
      sysreg = new SysReg[MAX_PROCESS];
      sliceCPU = 20; /* slice (квота), при которой
      полностью выполняются программы не превышающие
      20 блоков/строк в многозадачном режиме */
      job = NULL; // очередь заданий пуста
      this -> statistic = new Statistic;
      // метод Multi Processing
      this -> quotaProcess = -1;
      this -> j = 0;
    }
    // конструктор прототиптрования
    Scheduler(Scheduler *scheduler){
      this ->cpu = scheduler->cpu;
      this->MAX_PROCESS =  scheduler->MAX_PROCESS;
      this-> sysreg =  scheduler->sysreg;
      this-> sliceCPU = scheduler-> sliceCPU;
      this-> job = scheduler->job;
      this -> statistic = scheduler->statistic;
    }

    Interrupt execute (Job *job, MMU * mmu){
      Interrupt interrupt; /* OK, Error, Exit = 16,
            Sys, Div_0, Timer, Lan = 32, Dev = 254 */
      ProcessImage * processImage = NULL;
      // установить PC для процессов
      for (int i = 0; i < MAX_PROCESS;i++){
        /* -1 задание отсутствует, т.е. реальных
           процессов меньше, чем MAX_PROCESS */
        if (job[i].getAddr() != -1) {
          processImage = job[i].getProcessImage();
          if (processImage == NULL) { /* задание без
                              processImage, lab 1, 2 */
  sysreg[job[i].getId()].setState(PC,job[i].getAddr());
          }
        }
        // начало программы в памяти
      }
```

```
      while (true) {
        // увеличить счетчик времени на цикл CPU

        Timer::tick();
        for (int i = 0; i < MAX_PROCESS;i++){
          // -1 задание отсутствует
          if (job[i].getAddr() == -1)  continue;
          // ресурсы процесса
          if (job[i].getDone()){
            processImage = job[i].getProcessImage();
            // задание не с processImage
            if (processImage == NULL)
            interrupt= cpu->exeInstr(job[i].getAddr(),
                          &sysreg[job[i].getId()]);
            else if (processImage != NULL){/* задание
              с processImage */
              // -1 по умолчанию
              if (processImage->getTbegin()< 0)
            processImage->setTbegin(Timer::getTime());
              int AddrPC= processImage->getSysReg()->
                                        getState(PC);
              interrupt = cpu->exeInstr(AddrPC,
                          &sysreg[job[i].getId()]);
              // увеличить Texe
              processImage->setTexec(processImage->
                    getTexec()+1); // время выполнения

          /* метод RR только если установлена  квота.
             По умолчанию -1. То же, если установлен
             массив приоритетов с квотой DP */
              if (processImage->getTimeSlice() > 0) {
      // время выполнения процесса соответствует квоте
                if (processImage->getTexec() %
                  processImage->getTimeSlice()==0) {
                  // 1. убрать процесс из задания
                  job[i].setDone(false);
                  // 2. вытеснить процесс
                  preemptive(processImage,mmu);
                }
              }
/* выполнение вытеснения процесса по установленным
параметрам, каждому параметру соответствует метод, если
несколько параметров, то первый в векторе SRT,HRRN */
              paramPreemptive(processImage, mmu);
            }
           if(interrupt == Exit){ // процесс завершен
```

```
            job[i].setDone(false);// задание выполнено
            processImage = job[i].getProcessImage();
            // задание с processImage
            if (processImage != NULL){
              //время завершения
              processImage->setTterminate(
                                Timer::getTime()+1);
              // очистить память
              mmu ->swapOut(processImage);
/* удаляем процесс из очереди Running и ставим в
очередь ExitProcess в следующем  вызове dispatch()
отмечаем процесс как завершенный, информация о
процессе в PCB */
              processImage->setState(ExitProcess);
              }
              // +1, еще будет находиться в очереди
            }
            if ((interrupt == Error) ||
                              (interrupt == Empty)) {
          cout << "interrupt == "<< interrupt <<endl;
            // результат не предсказуем;
            job[i].Debug();
            return interrupt;
           }
         }
       }
       // все процессы завершены
       if(!isJob(job)) return OK;
     }
     return OK;
   }

 // проверка есть ли в задании процесс для выполнения
    bool isJob (Job *job){
      bool isJob = false;
      for (int i = 0; i < MAX_PROCESS;i++){
        // -1 задание отсутствует
        if (job[i].getAddr() == -1)  continue;
        if (job[i].getDone()) isJob = true;
      }
      return isJob;
    }

Interrupt execute(int addr, int id, CPU *cpu){
  // id идентификатор процесса
      // цикл исполнения программы CPU
      Interrupt interrupt;
```

```
      // начало программы в памяти
      sysreg[id].setState(PC,addr);
      SysReg *sysreg_ = &sysreg[id];
      for (int i=0; i <  sliceCPU; i++){
        interrupt = cpu->exeInstr(addr,sysreg_);
        if((interrupt == Exit) || (interrupt == Error)
          || (interrupt ==  Empty) ) return interrupt;
      } // end while
      return interrupt;
    }// end execute

    bool scheduleJob(MMU* mmu, bool priority){
       /* подготовить задания на выполнение,
          если не пустая очередь */
      DebugSPNQueue();
      // сформировать задание job для очереди Running
      // моделирование многозадачного режима
      job = new Job[MAX_PROCESS];
      ProcessImage * processImage = NULL;
    /* формируем задание для процессов < MAX_PROCESS
    1. из очерели Swapped ставим в очередь Running */
      int size_ = size(Swapped);
      for (int i = 0; i < size_;i++){
        processImage=(ProcessImage *)(front(Swapped));
        pop(Swapped);
        // 1.2. увеличить время обслуживания
        processImage->setTservice(processImage->
                                  getTservice()+1);
        push(Running, processImage);
      }
      // 2. сортируем
      if (vParam.getStatTime() != noParam )   {
        if (vParam.getStatTime() == TimeExecNotPreem)
          // по времени выполнения
          ProcessNext(TimeExec);
        else
          ProcessNext(vParam.getStatTime());
      }
      // упорядочить очередь по приоритету
      if (priority) {// метод DP
        // без параметров предсказуемого времени
        ProcessNext(noParam);
      }
/* подготовить задание и загрузить программы в память
   maxProcess == true превышение MAX_PROCESS,
   есть еще процессы maxProcess  == false,
   все процессы вошли в задание */
```

```
    size_ = size(Running);
    if (quotaProcess == -1){
      j = 0;
      for (int i = j; i < size_;i++){
        if ( j > MAX_PROCESS -1) return true;
        processImage=(ProcessImage *)(front(Running));
        mmu->swapIn(processImage);
        job[j].set(processImage->getAddr(),
             processImage->getID(),true,processImage);
        pop(Running);
        push(Running, processImage);
        j++;
      }
      return false;
    }
    else { // для многопроцессорного метода
      int k = j; // начало выдачи квоты
      for (int i = j; i < size_;i++){
       if(j>MAX_PROCESS-1) return false;// нет задач
       if(j>kquotaProcess-1) return true;//квота дана
       processImage=(ProcessImage *)(front(Running));
       mmu->swapIn(processImage);
       job[j].set(processImage->getAddr(),
             processImage->getID(),true,processImage);
        pop(Running);
        push(Running, processImage);
        j++;
      }
      j=0;  // задача выдана
      return false; // квота выдана, нет больше задач
    }
   } // end scheduleJob

   // выполнить задание
   Interrupt execute(MMU * mmu){
     return execute(job,mmu);}

   // применяется при многопроцессорном планировании
   // MultiProcessor scheduling (MP)
   void setJob (Job *job) {this->job = job;}
   Job * getJob () {return this->job;}

/* установить квоту для CPU, т. е. сколько циклов
выполнит CPU. Применяется для моделирования
многозадачного режима и передачи управления */
   void setSliceCPU(int sliceCPU){this->
                              sliceCPU = sliceCPU;}
```

```
/* получить идентификатор процесса, идентификатор процесса
соответствует незанятому индексу в пуле регистров */
    int getID(){
       for (int i = 0; i < MAX_PROCESS;i++){
         if (sysreg[i].getStatus()) return i;
       }
       return -1;  // все регистры заняты
    } //

    SysReg * getSysReg(int ID){return &sysreg[ID];}

// трассировка состояния регистров для одного блока
    void DebugBlock(int id, CPU *cpu){
      cpu->Debug(&sysreg[id]);
    }

    // трассировка состояния регистров
    void DebugSysReg(int id){sysreg[id].Debug();}

    void push(State state,
          ProcessImage * processImage){
      processQueue[state].push( processImage);
    }
    Process* pop(State state){
      if (processQueue[state].empty()) return NULL;
      Process* process = processQueue[state].front();
      // удалить элемент из очереди первый
      processQueue[state].pop();
      return process;
    }

    void pop(State state, ProcessImage * processImage){
      ProcessImage * processImage_ = NULL;
      for (int i=0;
        i < (int)processQueue[state].size(); i++){
        processImage_ =(ProcessImage *)
                                processQueue[state].front();
        // удалить элемент из очереди первый
        processQueue[state].pop();
        if (processImage_ != processImage)
          processQueue[state].push(processImage_);
      } /* перебираем до конца очереди, чтобы
           сохранить ее порядок */
    }
    Process * front(State state){
                  return processQueue[state].front();}
```

```
    bool empty(State state) {
                    return processQueue[state].empty();}
    int size(State state) {
              return processQueue[state].size();}

    void setObservation(ProcessImage * processImage){
    statistic->setObservation(processImage->getUser(),
              processImage->getTexec(),processImage->
              getTservice() );
    }

    void clearTpredict(){statistic->clearTpredict();}
    double getTpredict(string user,
                    StatTime time){
  return statistic->getTpredict(user,time);}

/* упорядочить очереди по предсказанному времени, в
    соответствии с параметром */
    void ProcessNext(StatTime time){
      //1. Running
      sortQueue(Running,time);
      //2. Blocked
      sortQueue(Blocked,time);
    }

    void sortQueue (State state, StatTime time) {
      int size_ = size(state);
      double wk = 0;
      // представляем очередь как массив
      valarray <ProcessImage*> vaProcess ( size_ );
      /* для сортировки очереди процессов по времени */
      valarray <double>   va ( size_ );
      ProcessImage * processImage = NULL;
      for (int i = 0 ;i < size_; i++) {
        processImage = (ProcessImage *)(front(state));
        vaProcess [i] = processImage;
        /* упорядочивать по наименьшему оставшемуся
           времени, не надо, т.к. метод FCFS */
        if (time == noParam ) // метод DP
          va[i] = processImage->getPriority();
        else
          va[i]=statistic->getTpredict(vaProcess[i]->
                                    getUser(),time);
        pop(state);
        push(state, processImage);
      }
      // сортировка простым выбором
      for (int i = 0; i < size_- 1 ; i++) {
```

```
      // просмотр массива, начиная со второго элемента
        for (int j = i+1; j < size_; j++){
/* 1. выбирается наименьший элемент по сравнению с
      первым */
          if ( va[j] < va[i] ) {
// 2. он меняется местами с первым элементом swap
            wк = va[j];
               processImage = vaProcess[j];
            va[j] = va[i];
               vaProcess[j] = vaProcess[i];
            va[i] = wк;
               vaProcess[i] = processImage;
          }
        } // end for
// 3. изменяется первый элемент, идем вправо
      } // end for

      for (int i = 0; i < size_ ;i++){
        pop(state);  // удаляем из очереди state
      // ставим упорядоченную очередь после сортировки
        push(state, vaProcess[i]);
      }
    } // end sortQueue

    void setTthreshold(StatTime time) {
      vParam.setTthreshold(time, statistic->
                              getTimeThreshold(time));
    }

    // выполнение процесса по методу SRT и HRRN
    void methodPreemptive(ProcessImage * processImage,
                          MMU * mmu, StatTime time){
      double Tpredict = statistic->
           getTpredict( processImage->getUser(),time);
      if (Tpredict == -1) return; // нет наблюдений
      if ( (Tpredict - processImage->getTexec()) >
                  vParam.getTthreshold(time)) return;
      /* 1. иначе убрать все процессы из задания,
         кроме данного */
      for (int i = 0; i < MAX_PROCESS;i++){
      /* -1 задание отсутствует, т.е. реальных
         процессов меньше, чем MAX_PROCESS */
        if (job[i].getAddr() != -1) {
          if ( job[i].getProcessImage()->getID() !=
               processImage->getID()){
            job[i].setDone(false); /* 1.1. убрать
                                      процесс из задания
```

```
            1.2. вытеснить все процессы, оставить
                 текущий processImage */
            preemptive(job[i].getProcessImage(),mmu);
          }
        }
      }
    }
// вытеснение в соответствии с заданными параметрами
    void paramPreemptive (ProcessImage *processImage,
                          MMU * mmu){
    // нет вытеснения SPN
      if (vParam.getState(TimeExecNotPreem)) return;
      // вытеснение процесса согласно параметру
      if (vParam.getStatTime() != noParam)
        methodPreemptive(processImage, mmu,
                                  vParam.getStatTime());
    }

    int getProcess(){return MAX_PROCESS;}
    void setQuotaProcess(int quotaProcess){this->
                          quotaProcess = quotaProcess;}

    void DebugQueue (State state){
      bool empty_ = empty(state);
      if (empty_) {
        cout <<" queue is empty " << state<< endl;
        return;
      }
      int size = processQueue[state].size();
      if (size == 0) return;
      ProcessImage * processImage =
          (ProcessImage *)processQueue[state].front();
          cout << endl;
          cout << "Queue  "<<processImage-> NameOf(state)<<
                                      " size: "<<size << endl;
          cout << "ID  =  "<<processImage->getID() << endl;
          cout << "PC  =  "<<processImage->getSysReg()->
                                        getState(PC) << endl;
          cout << "ProcessImage :" << endl;
      processImage->Debug();
      cout << endl;
    }
    void DebugSPNQueue (State state){
      if (empty(state)) {
        cout <<"        is empty " << endl;
        return;
      }
```

```
    int size_ = size(state);
    ProcessImage * processImage = NULL;
    for (int i = 0 ;i < size_; i++) {
      processImage = (ProcessImage *)(front(state));
      cout << endl;
      cout << "user = " << processImage->getUser()<< endl;
      if (vParam.getState(TimeExecNotPreem) ||
                        vParam.getState(TimeExec)){
        cout << "TpredictExec = " <<
        statistic->getTpredict(processImage->
                         getUser(),TimeExec)<< endl;
      }
     cout << "Texac = " << processImage->
                               getTexec() << endl;
      if (vParam.getState(TimeServ)) {
     cout << "TpredictServ = " << statistic->
     getTpredict(processImage->getUser(),TimeServ)<< endl;
     cout << "Tserv        = " << processImage->
                              getTservice() << endl;
      }
     cout << "Queue state  = " << processImage->
                                    getState()<< endl;
     cout << "Priority     = " << processImage->
                               getPriority() << endl;
     cout << "TimeSlice    = " << processImage->
                               getTimeSlice()<< endl;
      pop(state);
      push(state, processImage);
    }
  } // end DebugSPNQueue

  void DebugSPNQueue(){
    cout << "Method : queue order "<< endl;
    if (vParam.getState(TimeExec)) // метода SRT
        cout << "Tthreshold SRT = " <<
             vParam.getTthreshold(TimeExec) << endl;
    if (vParam.getState(TimeServ)) // метода SRT
      cout << "Tthreshold HRRN = " <<
             vParam.getTthreshold(TimeServ) << endl;

    cout <<" queue Running " << endl;
      DebugSPNQueue(Running);
    cout <<" queue Swapped " << endl;
      DebugSPNQueue(Swapped);
    cout <<" queue Blocked " << endl;
      DebugSPNQueue(Blocked);
    cout <<" queue ExitProcess " << endl;
```

```
        DebugSPNQueue(ExitProcess);
    }

    // вектор параметров для планирования
    VectorParam vParam;
  private:
    CPU *cpu;
    int MAX_PROCESS;
    Job* job;
    SysReg *sysreg;
    int sliceCPU;
/* очереди процессов в соответствии с их состояниями
   The queue class supports a first-in, first-service
   (FIFS, FIFO) method */
    queue <Process*> processQueue[NUMBER_OF_STATE];

    Statistic * statistic; // статистический расчет

/* распределение процессов по процессорам в
   методе Multi Processing */
    int quotaProcess;
    int j; // используются в подготовке задания
    // вытеснение процесса: метод RR и SRT, HRRN
    void preemptive(ProcessImage * processImage,
                                          MMU * mmu){
      ProcessImage * processImage_w = NULL;
      int size_ = size(Running);
      for (int i = 0; i < size_;i++){
      processImage_w=(ProcessImage *)(front(Running));
        if (processImage_w == processImage ){
          pop(Running);
      // 1. поставить процесс в очередь Swapped
          push(Swapped, processImage);
      // 2. освободить память swap, занятую программой
          mmu->swapOut(processImage);
      // 3. увеличить время обслуживания
          processImage->setTservice(processImage->
                                    getTservice()+1);
        } else {
          pop(Running);
          push(Running, processImage_w);
        }
      }
    }
};

#endif
```

Модуль Statistic.h

```
#ifndef __Statistic_H__
#define __Statistic_H__

#include "ArchitectureCPU.h"
#include "Process.h"
#include <queue>
#include <hash_map>

using namespace stdext; // for hash_map
using namespace System; // for Math::

// сохранение наблюдений по двум параметрам
class Table {
  public:
    Table(int tExec, int tServ){
      this->tExec = tExec;
      this->tServ = tServ;
    }
    int getTime(StatTime time) {
      if (time == TimeExec) return tExec;
      if (time == TimeServ) return tServ;
      return -1; // Error
    }
  private:
    int tExec; // время выполнения
    int tServ; // время обслуживания
};

class Statistic {
  public:
    Statistic( ){}

    void setObservation(string user, int tExec,
                                          int tServ){
      hmIter = ovserv.find(user);
      if ( hmIter == ovserv.end() ){
        //If no match is found, end() is returned
        vector <Table> tb;
        tb.push_back(Table(tExec,tServ));
        ovserv.insert(pairObserv (user, tb));
      }
      else { // tServ для метода HRRN
        hmIter ->second.push_back(Table(tExec,tServ));
      }
    }
```

```
// взвешенное среднее от 0 < alfa < 1,
double getTpredict(string user, StatTime time){
  hmIter = ovserv.find(user);
  if ( hmIter == ovserv.end() ) return -1.0;
  // число наблюдениий
  unsigned n = hmIter ->second.size();
  if ( n == 0 ) return -1;
  /* установлен взвешенный коэффициент для
     последнего наблюдения, т.к. с 0, то n-1 */
  double alfa = 0.8;
  double Tpredict = alfa * hmIter->
                        second[n-1].getTime(time);
/* просмотр с предпоследнего наблюдения до второго
  i > 0 все кроме последнего */
  for (unsigned i = n - 1; i > 0; i--) {
    Tpredict = Tpredict + Math::Pow((1-alfa),i) *
        alfa * hmIter->second[i-1].getTime(time);
  }
  return Tpredict;
}

double getTpredictSimple(string user,
                                    StatTime time){
  // обычное среднее не используется
  hmIter = ovserv.find(user);
  if ( hmIter == ovserv.end() ) return -1.0;
  // число наблюдениий
  unsigned n = hmIter ->second.size();
  double Tpredict = 0;
  // просматриваем наблюдения
  for (unsigned i = 0; i <n; i++) {
    Tpredict = Tpredict + hmIter ->
                        second[i].getTime(time);
  }
  return Tpredict/n; // делим на число наблюдений
}

void clearTpredict(){
  ovserv.erase(ovserv.begin(), ovserv.end());
  // очистить наблюдаемые данные
}

/* порог срабатывания с минимально
   предполагаемым временем в соответствии с
   параметром наблюдения и числом наблюдений */
double getTimeThreshold (StatTime time){
  int size_ = ovserv.size();
```

```
      // коэффициент, использующий сигмоидную функцию
      double beta = 1/(1 + Math::Pow(Math::E,-size_));
      hmIter = ovserv.begin();
      double TpredictMin = getTpredict(hmIter->first,
                                                time);
   /* 1. просматриваем все наблюдения процессов
     просмотр массива, начиная со второго элемента */
      for ( hmIter= hmIter++; hmIter != ovserv.end();
                                            hmIter++ ){
     // выбираем процесс с минимально предполагаемым временем
       if(TpredictMin > getTpredict(hmIter->first,time)){
          TpredictMin = getTpredict(hmIter->first,time);
      }
      }
      return TpredictMin*beta; // порог активации
    }

  private:
/* хранение наблюдений для вычисления предполагаемого
   времени по методу SPN */

    hash_map <string, vector<Table>> ovserv;
    hash_map <string, vector<Table>>::iterator hmIter;
    typedef pair <string, vector<Table>> pairObserv;
};

#endif
```

Модуль MPDispatcher.h

```
#ifndef __MPDispatcher_H__
#define __MPDispatcher_H__

#include <iostream>
using namespace std;
#include "ArchitectureCPU.h"
#include "VirtualMemory.h"
#include "Scheduler.h"
#include "Process.h"

using namespace System; // for Math::

/* MultiProcessor scheduling (MP) - многопроцессорное
планирование. Метод Global queue - планирование очередей.
Массив системных регистров для каждого процессора
учитывается в dispatch */
```

```
class MPDispatcher: public Dispatcher {
  public:
    MPDispatcher (int SIZE_OF_VIRTUAL_MEMORY_IN_IMAGES,
     Scheduler * scheduler,MMU * mmu,int MAX_PROCESSOR):
    Dispatcher(SIZE_OF_VIRTUAL_MEMORY_IN_IMAGES,
                                      scheduler,mmu) {
    this->MAX_PROCESSOR = MAX_PROCESSOR;
    vaScheduler=valarray <Scheduler *>(MAX_PROCESSOR);
        vaStatus    = valarray <bool> (MAX_PROCESSOR);
    for(int i=0; i < MAX_PROCESSOR; i++) {
      this->vaScheduler[i] = new Scheduler(scheduler);
      this-> vaStatus[i]= true; /* процессор свободен,
                                   false - занят */
    }
    }

/* сформировать задание, замещение scheduleProcess
   класса Dispatcher */
   void scheduleProcess(MMU * mmu, bool priority){
cout <<
" *** scheduleProcess - overriding functions *** "
<<endl;
/* 1. распределить процессы по процессорам; сколько
   процессов может выполняться на одном процессоре */
int quotaProcess=(int)Math::Round((double)scheduler->
                        getProcess()/MAX_PROCESSOR);
    scheduler->setQuotaProcess(quotaProcess);
    bool maxProcess = false;
      // определить, есть ли процессы в очереди
 for(int i=0; i < MAX_PROCESSOR; i++) {
  // 2. сформировать задание из общей очереди
  maxProcess =  scheduler->scheduleJob(mmu, priority);
  // 3. передать задание свободному процессору
  vaScheduler[i]->setJob(scheduler->getJob());
  // 4. отметить, что процессор занят
  vaStatus[i] = false;
  if (! maxProcess) break;
/* maxProcess  == true превышение MAX_PROCESS, есть
   еще процессы maxProcess  == false, все процессы
   вошли в задание */
  }
    MPDebug();
    mmu->DebugMemory();
    }
    // 3. выполнить задания на процессорах
   Interrupt executeProcess (MMU * mmu ){
    cout <<
```

```
    " *** executeProcess - overriding functions *** "
    <<endl;
    Interrupt interrupt;
    for(int i=0; i < MAX_PROCESSOR; i++) {
     if (! vaStatus[i]) {// занят
       interrupt = vaScheduler[i]->execute(mmu);
       vaStatus[i]= true; // освободить процессор
     }
    }
    return interrupt;
   }

   void MPDebug(){
    for(int i=0; i < MAX_PROCESSOR; i++) {
     cout <<
      "--- Processor --- " << i <<" state = " <<
      vaStatus[i] << endl;
     }
    }
  private:
    int MAX_PROCESSOR;
    valarray <Scheduler *> vaScheduler;
    valarray <bool> vaStatus;
};
#endif
```

Приложение С

Реализация категорий на языке программирования C++

Приложение состоит из пяти разделов. Каждый раздел посвящен описанию категории и ее реализации на языке программирования C++. Последовательно рассматриваются категории и их виды: агрегация — с. 178, наследование — с. 184, ассоциации — с. 207, использования — с. 215, конкретизации — с. 218.

Агрегация и наследование образуют два вида иерархии, поэтому они рассматриваются вначале. Синтаксис C++ для различных программных конструкций вводится по мере реализации категорий.

С.1. Категория агрегация

На основе отношения агрегация реализуется иерархия «Целое — Часть» (has-a). В зависимости от реализации могут быть три вида агрегации: по ссылке, по значению и вложением.

Рассмотрим категорию, в которой объекты класса A являются целым по отношению к объектам класса B, объект класса B является частью объекта класса A.

$$C_{\text{has-a}} = \frac{A \xrightarrow{\text{has-a}} B}{a \xrightarrow{g} b}.$$

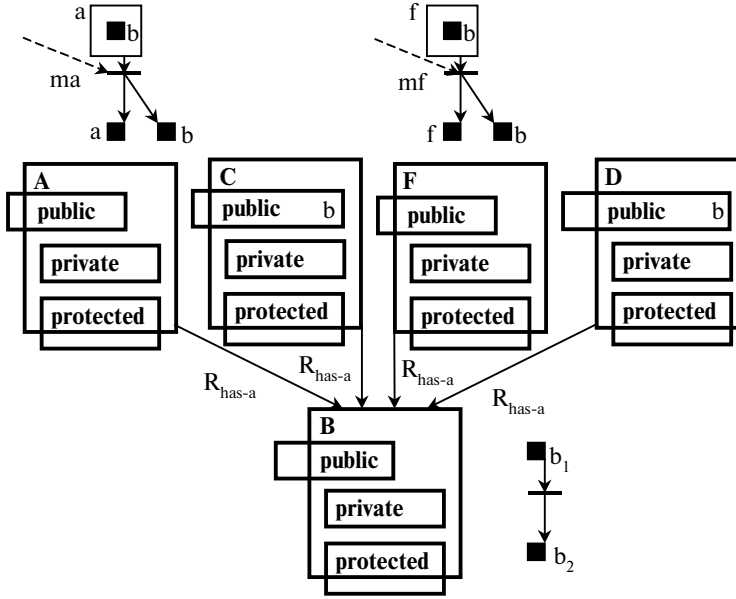


Рис. С1. Агрегация по ссылке (указателю)

Агрегация по ссылке. При агрегации по ссылке объект класса A перестает быть единым целым с объектом класса B в том смысле, что объект класса B можно создать или уничтожить независимо от объекта класса A (см. рис. С1).

В примере программы, демонстрирующей агрегацию по ссылке, приведены варианты связывания объектов: посредством конструктора, через общедоступную переменную, посредством метода, а также вызов метода объекта, являющегося частью, посредством метода объекта, являющегося целым. Они являются морфизмами g для категории $C_{\text{has-a}}$.

Кроме того, следующие морфизмы входят в категорию ma, mf.

При удалении объекта b нарушается целостность всех отношений в программе.

```
#include <iostream.h>
#include "string.h"

// aggregation агрегация по ссылке

class B; // Часть

// объекты являются "целым" по отношению к объекту B
class A{// объект связывается конструктором
  public:
    A(B &b)  { this -> b = &b;}
    B * ma() {return b;}; // доступ через метод к объекту
  private:
    B *b;
};

class C{// объект связывается конструктором
  public:
    C(B &b)  { this -> b = &b;}
    B *b;
};

class D{// объект связывается непосредственно по переменной
  public:
    D()  {}
    B *b;  // доступ через переменную к объекту
};

class F{// объект связывается посредством функции
  public:
    F()  {}
    void Link(B &b) { this -> b = &b;}
    B * mf() {return b;} // доступ через метод к объекту
  private:
    B *b;  // доступ через переменную к объекту
};

/* объект является частью по отношению к объектам
классов A,C,D,F */
class B{
  public:
    B(){ b=0;}
    int b;
    void mb(){
      b += 1;
```

```
      cout<<"variant link = "<< b << endl;
    }
};

void main( ){
  B b;
  A a(b); // 1 вариант связи объектов

  C c(b); // 2 вариант связи объектов
  D d;       // 3 объявление объекта
  d.b = &b; // 3 вариант связи объектов

  F f;        // 4 объявление объекта
  f.Link(b); // 4 вариант связи объектов

  // объект вызывает метод другого объекта.
  a.ma()-> mb(); //1
  c.b   -> mb(); //2
  d.b   -> mb(); //3
  f.mf()-> mb(); //4
}
```

На консоль выводятся следующие результаты:

variant link = 1
variant link = 2
variant link = 3
variant link = 4

Агрегация по значению. При агрегации по значению объект класса B не существует отдельно от объекта класса A (см. рис. С2). В категорию входят следующие морфизмы: ma, mf.

Пример программы, демонстрирующий агрегацию по значению. При удалении объекта b, объявленного с помощью конструктора в главной программе, отношение агрегации не изменится, так как этот объект не находится в отношении агрегации.

```
#include <iostream.h>
#include "string.h"

// aggregation агрегация по значению
// Часть
class B{
  public:
    B(){ b=0;}
    int b;
```

```
    void mb(){
        b+=1;
      cout<<" b="<<b<< endl;
    }
};
```

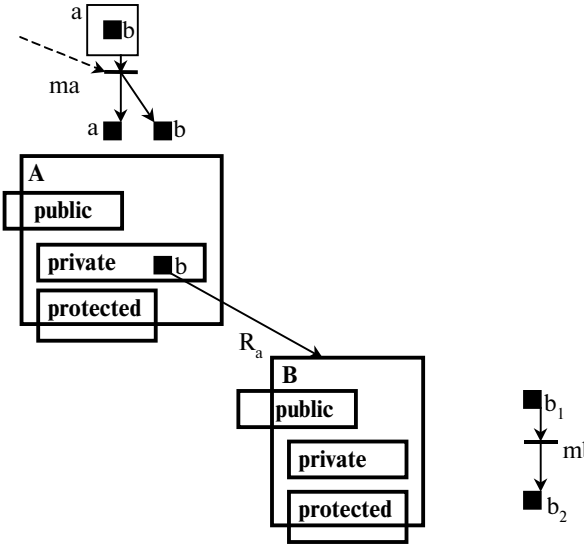


Рис. С2. Агрегация по значению

```
// Целое
class A{
  public:
    A();
    void ma(){
      b.mb();
    }
    B * magrb(){
      return &b;
    }

  private:
    B b; // агрегация
};

A::A(){}; // определение конструктора

void main( ){
  A a;
  B b;
```

```
  b.mb(); // объект b не является частью объекта a
  a.ma(); // объект b является частью объекта a
  a.magrb()->mb();
}
```

На консоль выводятся следующие результаты:

```
b=1
b=1
b=2
```

Агрегация вложением. При агрегации вложением класс и объект B определяется внутри класса и объекта A (см. рис. С3). В категорию входят следующие морфизмы: ma, mb, mbc.

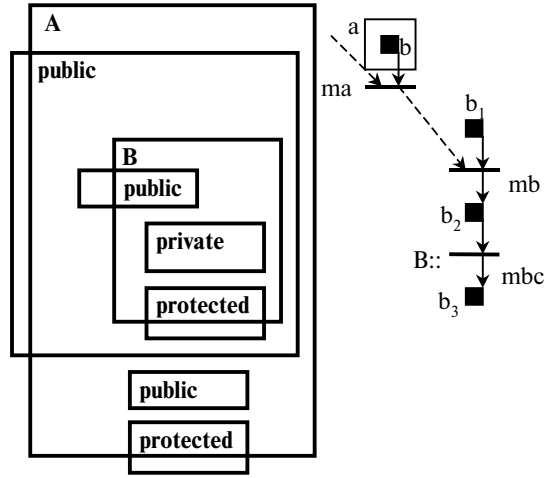


Рис. С3. Агрегация вложением

Пример программы, демонстрирующий агрегацию вложением. Объект b1 объявляется внутри класса A. Показан вариант использования статической (static) функции mbc.

Показан вариант объявления объекта b и применения метода mb.

Однако этот вариант не желателен, так как агрегация вложением создается с целью сокрытия объекта B или с целью его внутреннего использования в объекте A.

```
#include <iostream.h>
#include "string.h"
// aggregationNest агрегация вложением
// Целое
```

```
class A{
  public:
    // Часть
    class B{
      public:
        B(){ b=0;}
        int b;
        int mb(){
        b+=1;
          cout<<" b="<<b<< endl;
          return b;
        }
        static int mbc(int c) {
          cout<<"mbc()"<< endl;
          return c+1;
        }
    }; // end class B
    A();
    void ma(){
      cout<<"b1.mb()="<<b1.mb() << endl;;
      cout<<"const mbc(1)="<<B::mbc(1) << endl;;
    }
  private:
    B b1;
}; // end class A
A::A(){}; // определение конструктора
void main( ){
  A a;
  a.ma(); // объект b1 является частью объекта a
  cout<<"step 1"<< endl;
  A::B b;
  b.mb(); // объект b не является частью объекта a
}
```

На консоль выводятся следующие результаты:

```
b=1
b1.mb()=1
mbc()
const mbc(1)=2
step 1
b=1
```

С.2. Категория наследование

Категория наследование позволяет реализовать иерархию «Общее — частное»: $C_{is\text{-}a} = \dfrac{A \xrightarrow{is\text{-}a} B}{a \xrightarrow{g} b}$.

Основополагающим принципом разработки объектно-ориентированных программ является принцип подстановки, так как на основе данного принципа можно создавать полиморфные алгоритмы, которые не изменяются в процессе проектирования.

При анализе информационных систем необходимо учитывать этот принцип и применять те виды отношений наследования между объектами, которые реализуют данный принцип.

Принцип подстановки. Вместо объекта {a} суперкласса A можно *подставить* объекты {b,c, ...} подклассов {B,C,...} независимо от уровня иерархии и без видимых изменений поведения. Переменная a (указатель) обведена кружком и является полиморфной переменной для объектов {b, c, ...} (см. рис. С4).

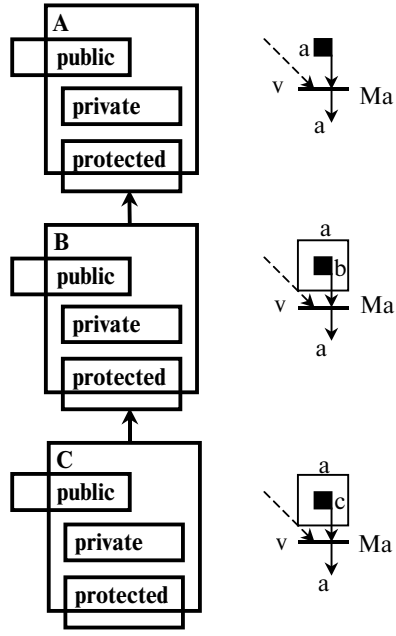


Рис. С4. Принцип подстановки

В категорию входит принцип подстановки, как основополагающий морфизм и морфизм замещения, например Ma.

Замещаемые методы Ma помечены маркером v. Далее приведен пример программы, демонстрирующий принцип подстановки, замещение методов, создание и разрушение объекта, утечку памяти, определение типа объекта.

```
#include "stdafx.h" // Microsoft
#include "string.h"
using namespace std;

// принцип подстановки
class A{ // суперкласс
  public:
    A(){a=0; cout<< "A a="<<a<<endl;}
    virtual ~A(){cout << "~A"<<endl;}
    virtual int Ma() // virtual для замещения функции
    {return a;}
  protected:
    int a;
};
class B:public A{ // подкласс
  public:
    B(){a=1; cout<< "B a="<<a<<endl;}
    virtual ~B(){cout << "~B"<<endl;}
    virtual int Ma() // замещение функции
    {return a;}
};

class C:public B{ // подкласс
  public:
    C(){a=2;cout<< "C a="<<a<<endl;}
    virtual ~C(){cout << "~C"<<endl;}
    virtual int Ma() // замещение функции
    {return a;}
};

int main() {

 A *a; // a – полиморфная переменная (указатель,ссылка)
 cout<<" создать объект A" <<endl;
 a = new A;
 cout<<a->Ma()<<endl;
 delete a; // удаление объекта

 cout<<" создать объект B " <<endl;
 a = new B;
 cout<<a->Ma()<<endl; // полиморфный вызов
 delete a;
//delete (B *)a; // если конструктор не virtual

 cout<<" создать объект C " <<endl;
 a = new C;
 cout<<a->Ma()<<endl;
```

```
 //C* c = dynamic_cast<C*>(a); // преобразование указателя
 //if (c != NULL){
 if (dynamic_cast<C*>(a) != NULL){
   cout<<" по указателю объект С " <<endl;
 }
 delete a;
//delete (C *) a; // если конструктор не virtual

 cout<<" вызов диструктора объекта с " <<endl;
 {
   C c;
 } // вызов диструктора объекта с

 cout<<" выход из программы " <<endl;
 return 0;
}
```

На консоль выводятся следующие результаты:

```
создать объект А
  A a=0
  0
  ~A
создать объект В
  A a=0
  B a=1
  1
  ~B
  ~A
создать объект С
  A a=0
  B a=1
  C a=2
  2
по указателю объект С
  ~C
  ~B
  ~A
вызов деструктора объекта с
  A a=0
  B a=1
  C a=2
  ~C
  ~B
  ~A
выход из программы
```

Рассмотрим разновидности категории наследования: расширение, специализацию, спецификацию, конструирование, обобщение, варьирование, ограничение, комбинирование, комбинирование через общих предков. Определим, для каких видов наследования выполняется *принцип подстановки*.

Расширение. В подобъект добавляются совершенно новые свойства, замещается по крайней мере один метод объекта суперкласса. Добавляются новые методы к объекту подкласса. Функциональные возможности менее крепко связаны с объектом суперкласса.

Так как функциональные возможности объекта суперкласса остаются неизмененными и доступными, то классы объектов являются подтипами, что не противоречит принципу подстановки. Однако принцип подстановки выполняется не полностью, так как добавляются новые методы к объектам подклассу (см. рис. С5).

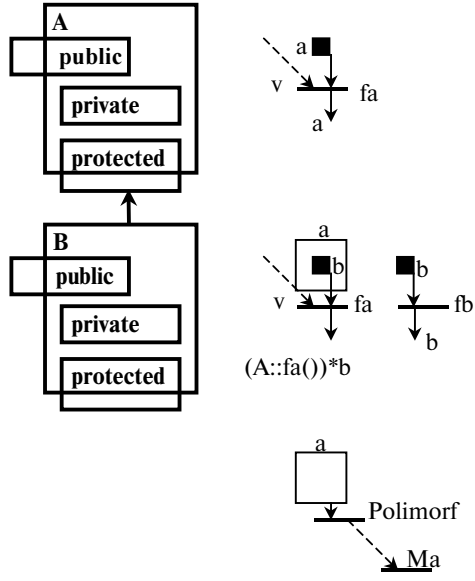


Рис. С5. Категория наследование: расширение

Методы fa и fb не изменяют состояния объектов, возвращают значения переменных соответственно a и b, что показано графически. Функция polimorf является общедоступной функцией с полиморфным параметром параметром объекта класса A. Далее приведен пример программы, демонстрирующий категорию наследование: расширение.

В категорию входят морфизмы: fa и fb.

```
#include<iostream.h>
#include "string.h"
// категория наследование: расширение
// принцип подстановки выполняется не полностью

class A{
  public:
    A(){a=1;}
    virtual int fa(){
      cout<<"a="<<a<<endl;
      return a;
    }
  private:
    int a;
};

class B: public A{
  public:
    B(){b=2;}
    int fa(){ // замещение
      cout<<"b="<<b<<endl;
      out<<"a*b="<<(A::fa())*b<<endl;
      return (A::fa())*b;
    }
    int fb(){ // расширение функциональных возможностей
      return b;
    }
  private:
    int b; // расширение по данным
};
// общедоступный метод
void polimorf(A *a){
  cout<<"polimorf begin"<<endl;
  a->fa();
  // Алгоритм
  cout<<"polimorf end"<<endl;
}

void main(){
  A *a;

  a=new A;
  a->fa();
  polimorf(a);

// подстановка
```

```
  a=new B;
  a->fa();
//  a->fb(); неполное выполнение принципа подстановки
  polimorf(a);
}
```

На консоль выводятся следующие результаты:

```
a=1
polimorf begin
a=1
polimorf end
b=2
a=1
a*b=2
a=1
polimorf begin
b=2
a=1
a*b=2
a=1
polimorf end
```

Специализация. Объект подкласса является более конкретным, частным или специализированным случаем для объекта суперкласса. Объект подкласса удовлетворяет спецификации объекта суперкласса. Принцип подстановки выполняется полностью. В рассматриваемом примере функция fa объекта b изменяет его состояние (см. рис. С6).

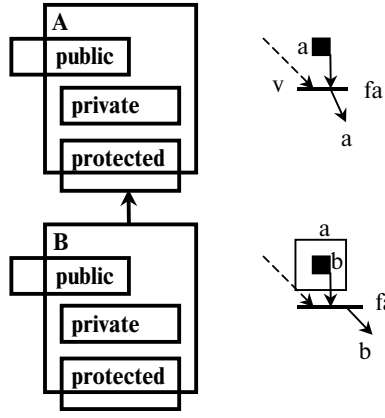


Рис. С6. Категория наследование: специализация

В категорию входит морфизм fa.

Далее приведен пример программы, демонстрирующий категорию наследование: специализация.

```
#include<iostream.h>
// категория наследование: специализация
  class A{
    public:
      A(){a=1;}
      virtual int fa(){ // замещение функции
        cout<<"a="<<a<<endl;
        return a;
      }
    private:
      int a;
  };

  class B: public A{
    public:
      B(){b=2;}
      int fa(){ // функция со специализированным поведением
        b+=1;
        cout<<"b="<<b<<endl;
        return b;
      }
    private:
      int b; // дополнительный атрибут
  };
void main(){
  A *a;

  a=new A;
  a->fa();
  a=new B; // принцип подстановки выполняется
  a->fa();
}
```

На консоль выводятся следующие результаты:

```
a=1
b=3
```

Спецификация. Спецификация гарантирует поддержку объектами определенного общего интерфейса. Объект суперкласса A описывает поведение, которое должно быть реализовано в объекте подкласса.

Нельзя создать объект от класса спецификации. Принцип подстановки выполняется между подклассами, сохраняющими данный интерфейс суперкласса {B,C, ...}. В рассматриваемом примере функция fa объекта b и с не изменяет состояние объектов (см. рис. С7).

В категорию входит морфизм fa, но отсутствует морфизм между $a \to b$.

Далее приведен пример программы, демонстрирующий вид категории наследование: спецификация.

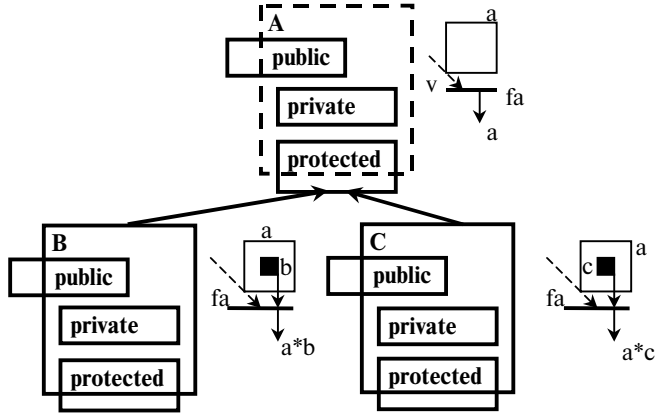


Рис. С7. Категория наследование: спецификация

```
#include<iostream.h>
// категория наследование: спецификация
// абстрактный класс
class A{
  public:
    A(){a=1;}
    virtual int fa()=0; // чисто виртуальная функция
  protected:
    int a;
};
class B: public A{
  public:
    B(){b=2;}
    int fa(){
      cout<<"a*b="<<a*b<<endl;
      return a*b;
    }
  private:
    int b;
};
class C: public A{
  public:
    C(){c=3;}
    int fa(){
      cout<<"a*c="<<a*c<<endl;
      return a*c;
    }
   private:
    int c;
};
```

```
void main(){
  A *a;

  // a=new A; нельзя создать экземпляр абстрактного класса

  a=new B;
  a->fa();

  a=new C;
  a->fa();
}
```

На консоль выводятся следующие результаты:

```
a*b=2
a*b=3
```

Конструирование. В подклассе изменяются имена методов, предоставляемые суперклассом, или модифицируются аргументы, объекты подкласса не являются подтипом суперкласса. Принцип подстановки не выполняется. В данном варианте область видимости объектов класса A при наследовании ограничивается, поэтому объект класса B не является подтипом. Принцип подстановки не выполняется (см. рис. С8).

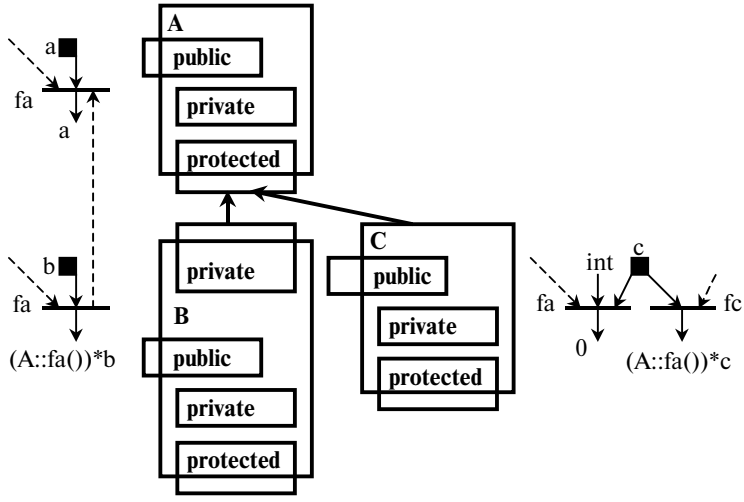


Рис. С8. Категория наследование: конструирование

В категорию входит морфизм fa, fc, но отсутствует морфизм для принципа подстановки.

Во втором варианте класс C является подклассом суперкласса A и метод fa(int c) объявляется как перегруженный.

Сценарий для различных вариантов взаимодействия объектов может быть получен объединением графов методов (см. на рисунке пунктирная линия со стрелкой).

Далее приведен пример программы, демонстрирующий категорию наследование: конструирование.

```
#include<iostream.h>
#include "string.h"

// категория наследование: конструирование
class A{
  public:
    A(){a=1;}
    int fa(){
      cout<<"a="<<a<<endl;
      return a;
    }
  private:
    int a;
};
class B: private A{
  public:
    B(){b=2;}
    void fb(){ b=3;}
    int fa(){
      cout<<"b="<<b<<endl;
      cout<<"a*b="<<(A::fa())*b<<endl;
      return (A::fa())*b;
    }
  private:
    int b;
};
class C: public A{
  public:
    C(){c=3;}
    int fa(int c){
      cout<<"c="<<c<<endl;
      return 0;
    }
    // изменение имени метода и его функциональных возможностей
    int fc(){
      cout<<"a*c="<<(A::fa())*c<<endl;
      return (A::fa())*c;
    }
  private:
    int c;
};
```

```
void main(){
  A *a;
  a=new A;
  a->fa();

  // a=new B; не реализуется принцип подстановки для класса B
  // b->fa(); операции не выполняются
  // b->fb();

  a=new C;
  // a->fa(9); // изменены аргументы метода
  // a->fc(); не реализуется принцип подстановки для класса B
  // операция не выполняется
}
```

На консоль выводится следующий результат:

a=1

Обобщение. Категория наследование обобщение противоположна специализации (перевернутая иерархия типов, которой следует избегать). Подкласс модифицирует либо переопределяет методы суперкласса с целью получения объекта более общей категории. Отношение этого вида желательно применять, когда подкласс основывается на значении, а не на поведении.

Методы суперкласса являются подмножеством области определения методов подкласса. Принцип подстановки выполняется, но так как функция суперкласса не должна использоваться, то будем говорить, что принцип постановки выполняется излишне полно (см. рис. С9).

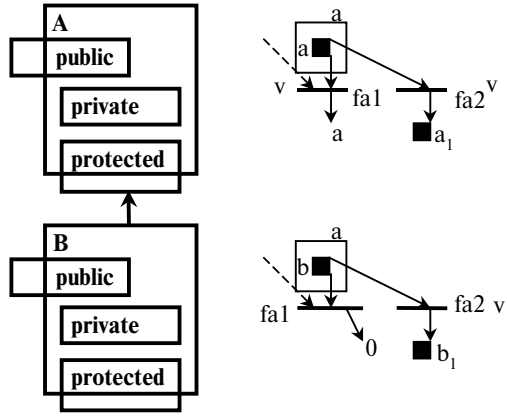


Рис. С9. Категория наследование: обобщение

В категорию входит морфизм fa1, fa2.

Далее приведен пример программы, демонстрирующий категорию наследование: обобщение.

```
#include<iostream.h>
// категория наследование: обобщение
class A{
  public:
    A(){a=1;}
    virtual int fa1(){
      cout<<"a="<<a<<endl;
      return a;
    }
    virtual void fa2(){
      a+=1*5 + 3; // выражение
      cout<<"a="<<a<<endl;
    }
  private:
    int a;
};

// класс B обобщает класс A
class B: public A{
  public:
    B(){b=1;}
    int fa1(){ // метод в дальнейшем больше не используется
      cout<<"no use = fa1 "<<endl;
      return 0;
    }
    void fa2(){ // функциональное обобщение
      b+=1; // обобщение выражения класса A
      cout<<"b="<<b<<endl;
    }
  private:
    int b;
};

void main(){
  A *a;

  a=new A;
  a->fa1();
  a->fa2();

  cout<<endl;
  a=new B;
  a->fa1();
  a->fa2();
}
```

На консоль выводятся следующие результаты:

```
a=1
a=9
no use = fa1
b=2
```

Варьирование. Два класса имеют сходную реализацию, но не имеют иерархической связи между абстрактными понятиями. Отметим, что метод fa1() замещен в классе B, а метод fa2() не замещен.

Далее приведен пример программы, демонстрирующий категорию наследование: варьирование. Принцип подстановки выполняется не полностью для попарно рассмотренных вариантов.

В программе показан вариант, получаемый в результате варьирования. В тексте программы он находится в комментариях.

Рассматривается двойственность категории.

```
#include<iostream.h>

class A{ // суперкласс
  public:
    A(){a=1;}
    virtual int fa1(){
      cout <<"a="<<a<<endl;
      return a;
    }
    int fa2(){
      a+=10;
      cout <<"a="<<a<<endl;
      return a;
    }
  private:
    int a;
};

class B: public A{ // подкласс
  public:
    B(){b=2;}
    int fb1(){
      cout<<"b="<<b<<endl;
      return b;
    }
  private:
    int b;
};
```

```
/* альтернативный вариант варьирования
class B { // суперкласс
  public:
    B(){b=2;}
    virtual int fb1(){
      cout<<"b="<<b<<endl;
      return b;
    }
  private:
    int b;
};

class A: public B{ // подкласс
  public:
    A(){a=1;}
    int fa1(){
      cout <<"a="<<a<<endl;
      return a;
    }
    int fa2(){
      a+=10;
      cout <<"a="<<a<<endl;
      return a;
    }
  private:
    int a;
};
*/

void main(){
  A *a;

  a=new A; // подстановка выполняется
  a->fa1();
  a->fa2();

  cout<<endl;

  a=new B;
  a->fa1();
  a->fa2();
}
```

На консоль выводятся следующие результаты:

```
a=1
a=11
b=2
a=11
```

Ограничение. Возможности подкласса более ограничены, чем суперкласса. Категория наследование применяется при построении подкласса на основе уже имеющейся иерархии, которая не должна или не может быть изменена. Принцип подстановки не выполняется (см. рис. С10). В категории отсутствует морфизм подстановки, определены морфизмы fa и fb.

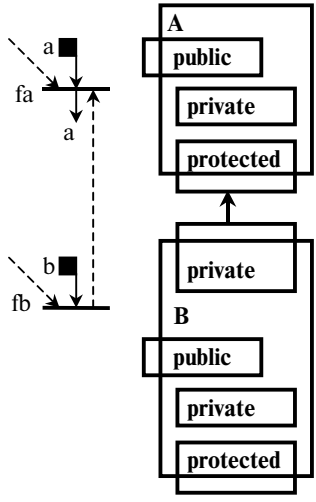


Рис. С10. Категория наследование: ограничение

Далее приведен пример программы, демонстрирующий категорию наследование: ограничение.

```
#include<iostream.h>
// категория наследование: ограничение
class A{
  public:
    A(){ a=1;}
    int fa(){
      cout<<"a="<<a<<endl;
      return a;
    }
  private:
    int a;
};
class B: private A{
  public:
    B(){b=2;}
```

```
    void fb(){
      b =2;
      cout<<"b= "<< b <<endl;;
    }
    int fa(){
      cout<<"a*b="<<(A::fa())*b<<endl;
      return (A::fa())*b;
    }
  private:
    int b;
};

void main(){
  A *a;
  B *b;

  a = new A;
  a->fa();
  // b = new B; принцип подстановки не выполняется
  // b->fa();
  cout << endl;

  b = new B;
  b –>fa();
  b –>fb();
}
```

На консоль выводятся следующие результаты:

```
a=1
a=1
a*b=2
a=1
b= 2
```

Комбинирование. Подкласс наследует черты более чем одного суперкласса. Реализация множественного наследования. Для рассматриваемого примера приводится два вида множественного наследования: посредством скрытого суперкласса A и общедоступного суперкласса B определяется подкласс C и посредством общедоступных суперклассов A и B определяется подкласс D. Для первого вида принцип подстановки не выполняется. Для второго вида принцип подстановки выполняется не полностью и только по ветки суперкласса A (см. рис. С11).

Категория основана на подкатегориях, представляющих собой уже рассмотренные категории наследования, в этом ее сложность. Агрегация подкатегорий осуществляется функтором композиции °. Таким образом можно агрегировать различные категории наследования.

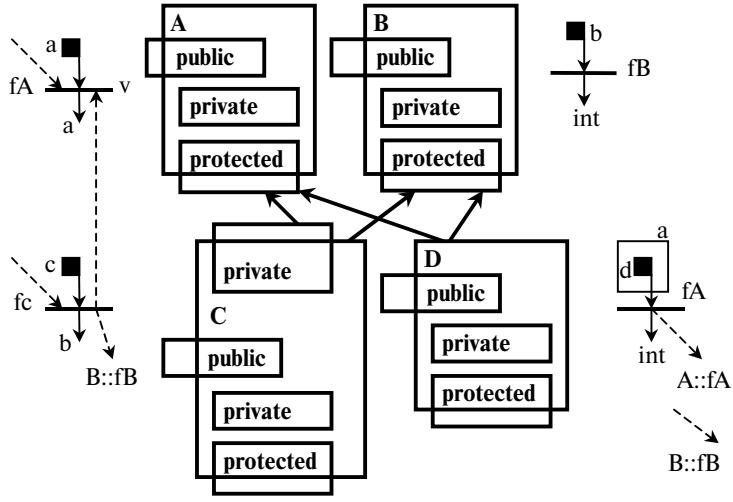


Рис. С11. Категория наследование: комбинирование

Далее приведен код программы, демонстрирующий вид категории наследование: комбинирование.

```
#include <iostream.h>
// combination – множественное наследование

class A{
  public:
    A(){a=1;}
    virtual int fA(){ // реализуется замещение в классе D
      cout<<"a="<<a<<endl;
      return a;
    }
  private:
    int a;
};

class B{
  public:
    B(){b=2;}
    int fB(){
      cout<<"b="<<b<<endl;
      return b;
    }
  private:
    int b;
};
```

```
/* такое множественное наследование не реализует
   принцип подстановки */
class C:private A,public B{
  public:
    C(){c=3;}
    int fc(){
      cout<<"a*b*c="<<(A::fA())*(B::fB()*c)<<endl;
      return (A::fA())*(B::fB()*c);
    }
  private:
    int c;
};

// множественное наследование реализует принцип подстановки
class D:public A, B{
  public:
    D(){d=4;}
    int fA(){ /* расширение по ветке A, или можно сделать
                 по ветке B */
      cout<<"a*b*d="<<(A::fA())*(B::fB()*d)<<endl;
      return (A::fA())*(B::fB()*d);
    }
  private:
    int d;
};

void main (void) {
  A *a;
  B *b;
  C *c;

  c = new C;
  c->fc();
  a = new A();
  // a = new C(); принцип подстановки не выполняется
  b = new C();
  // a = new B(); B не наследуется от A, так нельзя писать
  cout << endl;

  a -> fA();
  a = new D();
  a -> fA(); /* принцип подстановки выполняется
                только по ветке A */

  b -> fB();
  // b = new D(); // принцип подстановки не выполняется
}
```

На консоль выводятся следующие результаты:

```
a=1
b=2
a*b*c=6
```

```
a=1
b=2
a=1
a=1
b=2
a*b*d=8
a=1
b=2
b=2
```

Комбинирование через общих предков. Суперклассы {A,..,B} наследуются от общего предка — суперкласса D. Подкласс C наследует черты суперклассов {A,..,B}, реализуя множественное наследование.

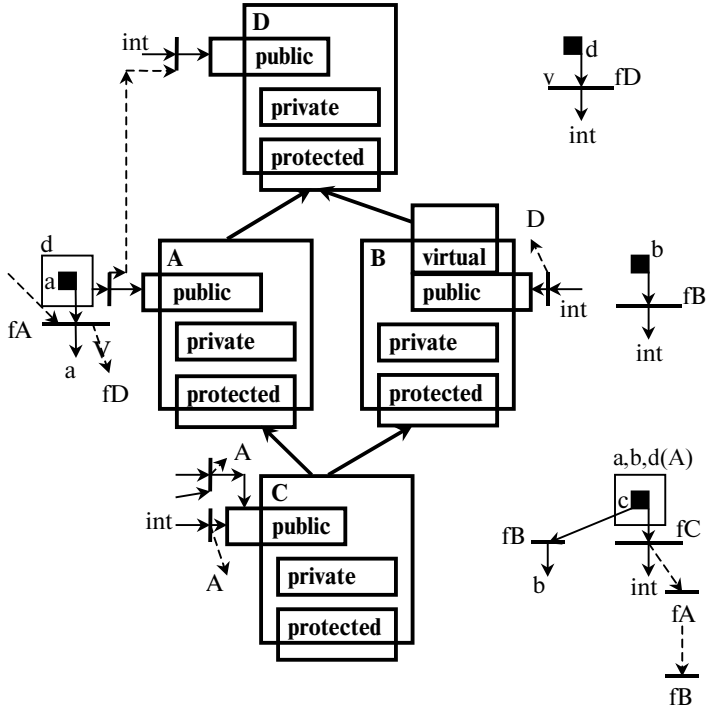


Рис. С12. Категория наследование: комбинирование через общих предков

В общем случае принцип подстановки может быть реализован полностью или частично. Для рассматриваемого примера полиморфная

переменная d реализует принцип подстановки для объектов классов D и A, а для объектов класса C через преобразование типа (указателя) к классу A (см. рис. C12).

Полиморфная переменная a реализует частично принцип подстановки для объектов классов A и C, так как в классе C реализована новая функция fC. Полиморфная переменная b реализует частично принцип подстановки для объектов классов B и C.

Категория, аналогичная комбинированию, основана на подкатегориях, представляющих собой уже рассмотренные категории наследования. Агрегация категорий осуществляется функтором композиции (, только исходным объектом, является один и тот же объект. Таким образом можно агрегировать различные категории наследования.

Далее приведен пример программы, демонстрирующий вид категории наследование — комбинирование через общих предков.

```
#include <iostream.h>
// combination —multiple-virtual, через общих предков
// общий предок — класс D
class D{
  public:
    D() {d=1; cout<<"D() d="<<d<<endl;}
    D(int d0){d=d0; cout<<"D(d) d="<<d<<endl;};
    virtual int fD(){
      cout<<"d->fD() ="<<d<<endl;
      return d;
    }
  private:
    int d;
};

/* виртуальные классы инициализируются лишь один раз
   вызовом безаргументного конструктора, если обращение
   к нему не осуществляется пользователем */
class A: //virtual D{ /* чтобы конструктор A вызывался один
                    раз надо по двум веткам ставить virtual,
                    посмотрите также работу virtual public */
  public D { /* выполняет конструкторы для класса D
                по умолчанию для каждой из ветвей */
  public:
    A(){a=2;}
    A(int d0):D(d0){cout<<"A"<<endl;}
    virtual int fA(){
      cout<<"a="<<a<<endl;
      return a*fD();
    }
```

```
  private:
    int a;
};

class B: virtual D{
  public:
    B(){b=3;}
    B(int d0):D(d0){cout<<"B"<<endl;}
    int fB(){
      cout<<"b->fB()="<<b<<endl;
      return b;
    }
  private:
    int b;
};

class C:public A,public B{
  public:
  C():B(),A(){c=4;}
  /* порядок следования конструкторов
     влияет на инициализацию */
  C(int c0):D(100),A(),B(){c=c0;}
  C(int c0,int c1):A(),B(),D(100){c=c0;cout<<"D3"<<endl;}
  /* разрешено использовать конструктор другого класса,
     который не является непосредственно предшествующим
     родителем конструктора класса D
     конструкторы для виртуальных базовых классов должны
     вызываться первыми */

  int fD(){
    cout<<"c->fD()="<<c<<endl;
    return c;
  }
  int fC(){
    c=fA()*fB()*c;
      cout<<"fA()*fB()*c="<<c<<endl;
    return c;
  }
  private:
    int c;
};

void main (void) {
  D *d;

  cout<<"step1"<<endl;
  d=new D;
  d->fD();
```

```
  d=new A; /* при public наследовании можно подставить
              экземпляр объекта */
  d->fD();

//d=new B; // при virtual наследовании нельзя подставить
           // экземпляр объекта
//d->fD();

//d= new C; нельзя сделать присвоение
  d= (A *)(new C(77)); // преобразование типов
  d->fD();

  cout<<"step2"<<endl;
  A *a;
  a = new A();
  a –>fD();
  a = new C(88); // преобразование типов
  a->fD();

  cout<<"step3"<<endl;
  B *b;
  b = new B();
//b –>fD(); недоступна, т. к. virtual наследование
  b = new C(22); // преобразование типов
  b->fB();

  cout<<"step4"<<endl;
  C c;
  c.fC();

  C c1(33);
  c1.fC();
}
```

На консоль выводятся следующие результаты:

```
step1
1 D() d=1
d->fD() =1
2 D() d=1
d->fD() =1
D(d) d=100
D() d=1
c->fD()=77
step2
D() d=1
d->fD() =1
```

```
D(d) d=100
D() d=1
c->fD()=88
step3
D() d=1
D(d) d=100
D() d=1
b->fB()=3
step4
D() d=1
D() d=1
a=2
c->fD()=4
b->fB()=3
fA()*fB()*c=96
D(d) d=100
D() d=1
a=2
c->fD()=33
b->fB()=3
fA()*fB()*c=6534
```

5.3. Категория ассоциация

Категория ассоциация основана на двухстороннем отношении, т. е. если объекты класса A и B связаны отношением ассоциации, то из объекта класса A доступен объект класса B, а из объекта класса B доступен объект класса A. Ассоциация может быть следующих видов: «один к одному» (1:1), «один ко многим» (1:N), «многие ко многим» (N:N):

$$C_{assoc}: \frac{A \xleftrightarrow{assoc} B}{a \xleftrightarrow{g} b}.$$

Ассоциация вида 1:1. Объекты классов A и B принадлежат категории ассоциация вида 1:1, если один объект класса A связан только с одним объектом класса B, а этот объект класса B связан с этим же объектом класса A (см. рис. С13).

В категорию входят морфизмы ma, M, getA, mb.

Рассмотрим пример программы, демонстрирующий категорию ассоциация вида 1:1. Вначале с помощью методов ma и mb объекты связываются. Через общедоступную переменную b объект класса A вызывается метод M объекта класса B. Через метод объекта класса B getA() возвращается ссылка на объект класса A и вызывается метод M объекта класса A.

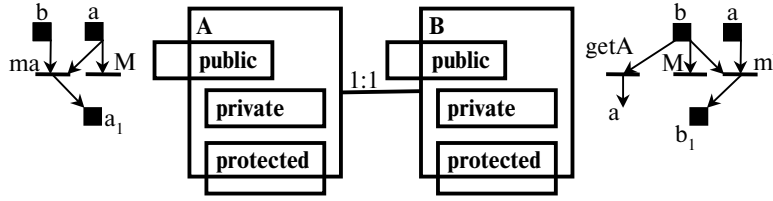


Рис. С13. Категория ассоциация 1:1

```
#include <iostream.h>
#include "string.h"

// association 1:1 ассоциация
class B;
class A{
  public:
    A();
    void ma(B &b){
      cout<<"link A=b" << endl;
      this->b =&b;
    }
    void M(){
      cout<<"MA" << endl;
    }
// protected:
  B *b;
};
A::A(){};

class B{
  public:
    B(){}
    void mb(A &a){
      cout<<"link B=a" << endl;
      this –>a=&a;
    }
    void M(){
      cout<<"MB" << endl;
    }
    A* getA(){
      return a;
    }
  protected:
    A* a;
};
```

```
void main(void){
  A a;
  B b;

  a.ma(b);
  b.mb(a);

  a.b->M();
  b.getA()->M();
}
```

На консоль выводятся следующие результаты:

```
link A=b
link B=a
MB
MA
```

Ассоциация вида 1:N. Объекты классов A и B принадлежат категории ассоциация вида 1:N, если один объект класса A связан с N объектами класса B, а каждый объект класса B связан с этим же объектами класса A.

Рассмотрим пример программы, демонстрирующий категорию ассоциация вида 1:N. Вначале объявляется объект класса A и несколько объектов класса B. Методы setA и setB связывают объекты (см. рис. С14).

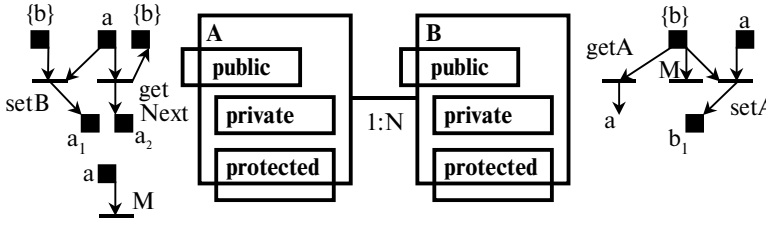


Рис. С14. Категория ассоциация 1:N

В категорию входят морфизмы 1:N, setB, getNext, M, getA, setA.

Метод объекта класса A — getNext позволяет последовательно просматривать объекты класса B, которые с ним связаны. К объектам класса B применяется метод M. Через метод объекта B — getA возвращается ссылка на объект A и вызывается метод M объекта A.

Методы setA, setB, getNext, getA могут выполняться последовательно несколько раз.

```
#include <iostream.h>
#include <string.h>
```

```
// association 1:N ассоциация
class B;
class A{
  public:
    A();
    int i,j; // количество элементов в массиве

    void setB(B &b0){
      cout<<"link A = "<< &b0 << endl;
      *b = &b0;
      cout<<"link A = "<< *b << endl;
      i+=1;
      b++;
    }

    B * getNext(){
      if (j < i )
      {
        j++;
        cout<<"getNext b1[j]="<< b1[j] << endl;
      return b1[j];
      }
      j=0;
      cout<<"getNext b1[j]="<< b1[j] << endl;
      return b1[j]; //
    }
    void M(){
      cout<<"MA" << endl;
    }
 // protected:
    B **b;
  private:
    B *b1[5];
};
A::A(){b=b1;i=-1;j=-1;};

class B{
  public:
    B(){}
    void setA(A &a){
      cout<<"link B=a" << endl;
      this ->a=&a;
    }
    void M(){
      cout<<"MB" << endl;
    }
    A* getA(){return a;}
```

```
protected:
  A* a;
};
void main(void){
  A a;
  B b,b1,b2;
// ассоциация 1:N двухсторонняя категория
  b.setA(a);
  b1.setA(a);
  b2.setA(a);
//
  a.setB(b);
  a.setB(b1);
  a.setB(b2);
// обращение из объекта "a" к объекту "b"
  a.getNext()->M();
  a.getNext()->M();
  a.getNext()->M();
  a.getNext()->M();
// обращение из объекта "b" к объекту "a"
  b.getA()->M();
  b1.getA()->M();
  b2.getA()->M();
}
```

На консоль выводятся следующие результаты:

```
link B=a
link B=a
link B=a
link A = 0x0065FDD4
link A = 0x0065FDD4
link A = 0x0065FDD0
link A = 0x0065FDD0
link A = 0x0065FDCC
link A = 0x0065FDCC
getNext b1[j]=0x0065FDD4
MB
getNext b1[j]=0x0065FDD0
MB
getNext b1[j]=0x0065FDCC
MB
getNext b1[j]=0x0065FDD4
MB
MA
MA
MA
```

Ассоциация вида N:N. Объекты классов A и B принадлежат категории ассоциация вида N:N, если каждый из N объектов класса A связан с N объектами класса B, а каждый из N объектов класса B связан с N объектами класса A (см. рис. С15).

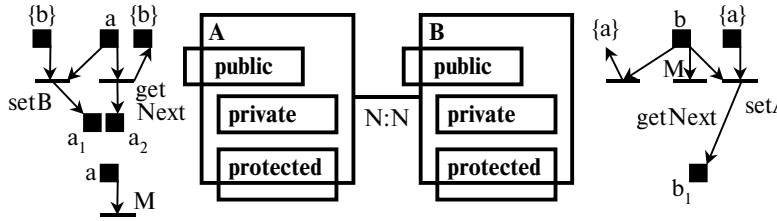


Рис. С15. Категория ассоциация N:N

В категорию входят морфизмы N:N, setB, getNext, M, getA, setA.

Рассмотрим пример программы, демонстрирующий категорию ассоциация вида N:N. Вначале объявляются объекты класса A и несколько объектов класса B. Методы setA и setB связывают объекты класса A с объектами класса B, и наоборот. Метод getNext класса A применяется к нескольким объектам, последовательно возвращая объекты класса B. К объектам класса B применяется метод M. Метод getNext класса B применяется к нескольким объектам, возвращая объекты класса A, к ним применяется метод M класса A.

Методы setA, setB, getNext выполняются последовательно несколько раз.

```
#include <iostream.h>
#include <string.h>
// association N:N ассоциация
class B;
class A{
  public:
    A();
    int i,j; // количество элементов в массиве

    void setB(B &b0){
      cout<<"link A = "<< &b0 << endl;
      *b = &b0;
      i+=1;
      b++;
    }
    B * getNext(){
      if (j < i )
      {
```

```
      j++;
      cout<<"getNext b1[j]="<< b1[j] << endl;
      return b1[j];
      }
      j=0;
      cout<<"getNext b1[j]="<< b1[j] << endl;
      return b1[j]; //
    }
    void M(){
    cout<<"MA" << endl;
  }
//protected:
    B **b;
  private:
    B *b1[5];
};
A::A(){b=b1;i=-1;j=-1;};
class B{
  public:
    int i,j;
    B(){a=a1;i=-1;j=-1;};
    void setA(A &a0){
      cout<<"link B=" <<&a<< endl;
      *a=&a0;
      i+=1;
      a++;
    }
    A * getNext(){
      if (j < i )
      {
        j++;
        cout<<"getNext a1[j]="<< a1[j] << endl;
        return a1[j];
      }
      j=0;
      cout<<"getNext a1[j]="<< a1[j] << endl;
      return a1[j];
    }
    void M(){
      cout<<"MB" << endl;
    }
  protected:
    A **a;
  private:
    A *a1[5];
};
void main(void){
  A a,a1;
  B b,b1;
```

```
// ассоциация N:N двухсторонняя категория
  b.setA(a);
  b.setA(a1);
  b1.setA(a);
  b1.setA(a1);
//
  a.setB(b);
  a.setB(b1);
  a1.setB(b);
  a1.setB(b1);
// обращение из объектов "a" к объектам "b"
  a.getNext()->M();
  a.getNext()->M();
  a1.getNext()->M();
  a1.getNext()->M();
//a.getNext()->M();
// обращение из объектов "b" к объектам "a"
  b.getNext()->M();
  b.getNext()->M();
  b1.getNext()->M();
  b1.getNext()->M();
}
```

На консоль выводятся следующие результаты:

```
link B=0x0065FDA0
link B=0x0065FDA0
link B=0x0065FD80
link B=0x0065FD80
link A = 0x0065FD98
link A = 0x0065FD78
link A = 0x0065FD98
link A = 0x0065FD78
getNext b1[j]=0x0065FD98
MB
getNext b1[j]=0x0065FD78
MB
getNext b1[j]=0x0065FD98
MB
getNext b1[j]=0x0065FD78
MB
getNext a1[j]=0x0065FDD8
MA
getNext a1[j]=0x0065FDB8
MA
getNext a1[j]=0x0065FDD8
MA
getNext a1[j]=0x0065FDB8
MA
```

С.4. Категория использование

Категория использование может быть двух видов: клиент-сервер и общность (друг — friend).

$$C_{use}: \frac{A \xrightarrow{use} B}{a \xrightarrow{g} b}.$$

Использование клиент-сервер. Объекты классов принадлежат категории, если объект класса A является сервером по отношению к объекту B. В качестве параметра для метода mb объекта b передается объект (или ссылка) класса A. Для контроля изменения значений переменных объектов применяются методы geta и getb (см. рис. С16).

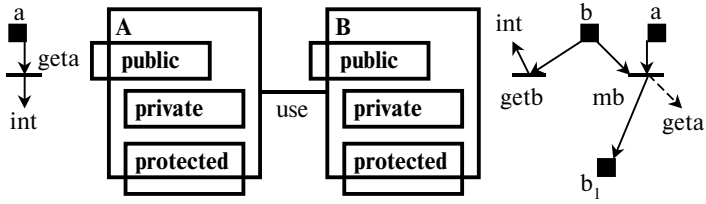


Рис. С16. Категория использования клиент-сервер

В категорию входит морфизм getb, на уровне $A \to B$, а не на уровне g, для g морфизмы geta и getb.

Рассмотрим пример программы, демонстрирующий категорию использования: клиент-сервер.

```
#include <iostream.h>
#include <string.h>
// категория использования client-server (клиент-сервер)
// server
class A{
  public:
    A():a(10){}
    int geta(){return a;};
  private:
    int a;
};
class B{ // client
  public:
    int b;
    B():b(0){}
    void mb(A &a){
      b=a.geta() + b;
    }
    int getb(){return b;};
};
```

```
void main(){
  A a;
  B b;

  cout<<" a.geta()="<<a.geta()<< endl;
  cout<<" b.getb()="<<b.getb()<< endl;

  b.mb(a); /* категория client-server (клиент-сервер)
              между объектами */
  cout<<" a.geta()="<<a.geta()<< endl;
  cout<<" b.getb()="<<b.getb()<< endl;
}
```

На консоль выводятся следующие результаты:

```
a.geta()=10
b.getb()=0
a.geta()=10
b.getb()=10
```

Использование общность. Объект класса A является другом (f — friend) по отношению к объекту класса B, если скрытые (p – private) переменные и методы, объявленные как f, доступны из объекта класса B (см. рис. С17).

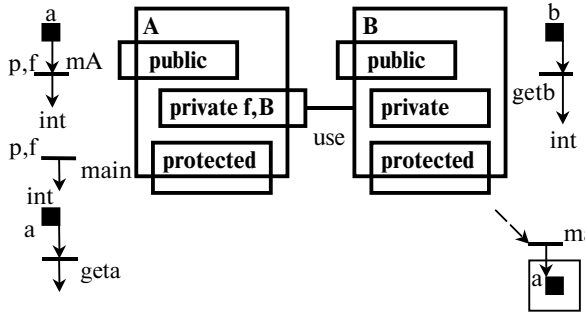


Рис. С17. Категория использования общности

В категорию входит морфизм friend на уровне A → B.

Метод объекта b или общедоступные методы, использующие скрытые (private) переменные или методы из объекта a, необходимо объявить как дружественные (**friend**) в объекте a. В этом случае мы говорим, что объект a находится в отношении использования к объекту b.

Пример программы, демонстрирующий категорию использование общность.

```
# include <iostream.h>
/* категория использования friend (общность) демонстрирует
   использование скрытого конструктора, внешнего метода,
   доступа к скрытым переменным из другого класса; объявление
   *friend class B* в классе A говорит о том, что на эти
   переменные можно сослаться из класса B, даже если
   они *** скрытые ***, например int a. */

class A {
  private:
    friend void mA(); /* должна быть friend, так как
                         вызывает скрытый конструктор */
    friend class B; // может быть объявлена в области public
    friend int main(); /* позволяет в main вызвать
                          — скрытый — конструктор */
    int a;
    A():a(500){}; // — скрытый — конструктор
  public:
  // friend class B;
    int geta(){return a;};
};

// общедоступная ссылка на класс A
A *aa;

// общедоступная функция
void mA(){
  aa=new A; // вызов — скрытого — конструктора
};

class B {
  private:
    int b;
    int d;
  public:
  /* для того чтобы в классе B была видна
     скрытая переменная — a —, в классе A определяем
     класс B как friend */
  B(A *a0) : b(a0->a) {} /* constructor инициализирует
                            значение b по значению a класса A */
  int getb(){return b;}
};

int main(){ // функция main перегружена
  mA();// создает объект со скрытым конструктором
       // должен быть friend класса A
  cout<<" aa->geta()= "<<aa->geta()<<endl;
```

```
  /* демонстрация использования скрытого конструктора A,
     т. к. конструктор A виден в main */

  B b(new A);
  cout<<" b.getb()= "<<b.getb() <<endl;

  return 0;
}
```

На консоль выводятся следующие результаты:

```
aa->geta()= 500
b.getb()= 500
```

С.5. Категория конкретизация

Категория конкретизация может быть двух видов: конкретизация функций и конкретизация классов.

$$C_{\text{template}} = \frac{A \xrightarrow{\text{template}} B}{a \xrightarrow{g} b}.$$

Конкретизация функций. Класс A конкретизирует функцию, если функция определена как шаблон (template) с параметром или параметрами T. В примере функция swap имеет два входных параметра (T), которые должны быть настроены. Рассмотрены два случая применения функции (swap) (см. рис. С18).

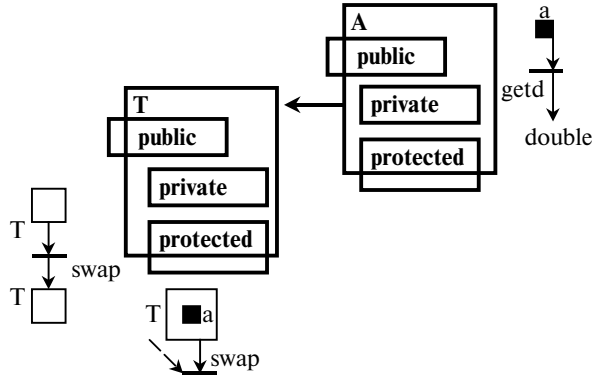


Рис. С18. Категория конкретизация функции

В категорию входит морфизм template на уровне $A \to B$, на уровне g морфизмы getd, swap.

В первом случае в качестве настраиваемых входных параметров T в функцию (swap) передаются целые значения i,j. Во втором случае в ка-

честве настраиваемых входных параметров T в функцию (swap) передаются объекты класса A. Методы, выводящие на печать значения данных, на графе опущены.

Далее приведен пример программы, демонстрирующий категорию конкретизации функции.

```
#include <iostream.h>
#include "string.h"

// template конкретизация функции

template <class T>
void swap (T& x, T& y) { /* берутся ссылки на экземпляры
                            объектов */
  T temp;
  temp = x;
  x=y;
  y=temp;
}

class A {
  public:
    int a,a1;
    A(){a=0;}
    A(int a,int a1, double d ) {
      this –>a=a; this –>a1=a1; this –>d=d;
    }
    double getd(){return d;}
    private:
    double d;
};

void main(void) {
  int i=2,j=33;

  cout << "i =" <<i<<" j ="<<j <<endl;
  swap (i,j);
  cout << " swap :i =" <<i<<" j ="<<j <<endl;

  A a(11,22,1.11);
  A b(33,44,2.22);
  cout <<"a.a =" <<a.a<<" a.getd()= "<<a.getd()<<" b.a =
                  "<<b.a <<" b.getd() = "<<b.getd()<<endl;
  cout <<"a.a1=" <<a.a1<<" b.a1 ="<<b.a1<<endl;
  swap (a,b);
  cout << " swap: a.a =" <<a.a <<" a.getd()= "<<a.getd()<<"
              b.a ="<<b.a <<"b.getd()= "<<b.getd() <<endl;
  cout << " swap: a.a1 =" <<a.a1<<" b.a1="<<b.a1<<endl;
}
```

На консоль выводятся следующие результаты:

```
i =2 j =33
swap :i =33 j =2
a.a =11 a.getd()= 1.11 b.a =33 b.getd()= 2.22
a.a1=22 b.a1 =44
swap: a.a =33 a.getd()= 2.22 b.a =11 b.getd()= 1.11
swap: a.a1 =44 b.a1=22
```

Конкретизация класса. Класс I конкретизирует класс A, если класс A определен как шаблон (template) с параметрами T. В качестве входных параметров объекта класса A передается объект класса I (см. рис. С19).

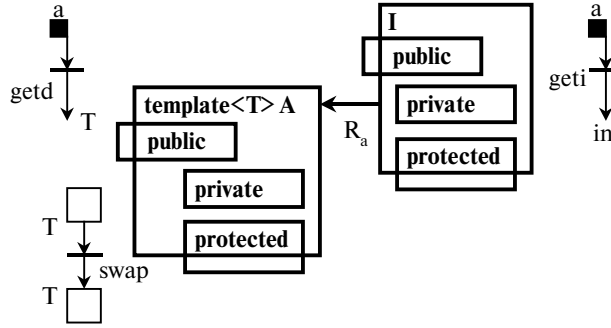


Рис. С19. Категория конкретизация класса

В категорию входит морфизм template на уровне $A \rightarrow B$, на уровне g морфизмы geti, getd, swap.

Далее приводится пример программы, демонстрирующий категорию конкретизация классов.

```
#include <iostream.h>
#include "string.h"

/* template конкретизация класса
   в шаблоне применяется отношение агрегации
   пример демонстрирует пересылку внешним и внутренним для A
   экземпляром объекта
   пересылаемые объекты должны быть одного и того же типа I */
template <class T>
class A {
  public:
    int a,a1;
    A(){a=0;}
    A(int a,int a1)  {
      this ->a=a; this ->a1=a1;
    }
```

```
    T & getd(){return d;}
    //
    void swap (T& x) { // берутся ссылки на объекты
      T temp;
      temp = x;
      x=d;
      d=temp;
    }

  private:
    T d; // агрегация по значению Ra
};
class I {
  public:
    I(){i=0;}
    I(int i){this ->i=i;}
    int geti(){return i;}
  private:
    int i;
};

void main(void) {

  I i(10);
  A <I> a(11,22);

  cout << " i.geti() = "<<i.geti()<<endl;
  cout << " a.getd().d.geti() = "<<a.getd().geti()<<endl;

  a.swap (i); /* замена экземпляра объекта, находящегося
                 внутри A на объект i */

  cout << "swap: i.geti() = "<<i.geti()<<endl;
  cout << "swap: a.getd().d.geti() = "<<a.getd().geti()<<endl;
}
```

На консоль выводятся следующие результаты:

```
i.geti() = 10
a.getd().d.geti() = 0
swap: i.geti() = 0
swap: a.getd().d.geti() = 10
```

СПИСОК ЛИТЕРАТУРЫ

1. *Dasgupta S.* Computer Architecture: A Modern Synthesis. Vol. 2: Advanced Topics. N. Y.: John Wiley & Sons, 1989.
2. *Семенов А. С.* Информационные технологии: объектно-ориентированное моделирование. Классификация моделей отношений и взаимодействий объектов с примерами на C++: учеб. пособие. М.: Изд-во МГТУ «Станкин», 2000.
3. *Тимоти Б.* Объектно-ориентированное программирование в действии: пер. с англ. СПб., 1997.
4. Что такое компьютерная архитектура. http://www-scm.tees.ac.uk/users/a.clements/arch/arch1a.htm
5. *Столингс В.* Операционные системы: пер. с англ. 4-е изд. М.: Изд. дом «Вильямс», 2004.
6. *Олифер В. Г.*, *Олифер Н. А.* Сетевые операционные системы. СПб.: Питер, 2001.
7. *Руссинович М.*, *Соломон Д.* Внутреннее устройство Windows: пер. с англ. 4-е изд. СПб.: Питер, 2006.
8. *Джонсон М.* Системное программирование в среде Windows: пер. с англ. 4-е изд. М.: Изд. дом «Вильямс», 2005.

СОДЕРЖАНИЕ

Семенов Александр Сергеевич

ПРОЕКТИРОВАНИЕ СЕТЕВЫХ ОПЕРАЦИОННЫХ СИСТЕМ
Практический курс

Книга издана в авторской редакции

Ответственный редактор *Н. Г. Карасева*
Технический редактор *П. С. Корсунская*
Корректор *О. А. Королева*
Компьютерная верстка: *А. А. Гуров*

Подписано в печать 12.08.08. Формат 60 x 84 /16
Печать офсетная. Бумага газетная.
Усл. печ. л. 13,02. Тираж 300 экз.

ЗАО «Издательское предприятие «Вузовская книга»
125993, Москва, А-80, ГСП-3, Волоколамское шоссе, д. 4.
МАИ, Главный административный корпус, к. 301а.
Т/ф (499) 158-02-35. E-mail: vbook@mail.ru; vbook@mai.ru
www.vuzkniga.ru