

Министерство образования и науки Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего профессионального образования  
«Комсомольский-на-Амуре государственный технический университет»  
Кафедра «Электропривод и автоматизация промышленных установок»

УТВЕРЖДАЮ  
Первый проректор ФГБОУ ВПО «КнАГТУ»  
А.Р. Куделько  
2011 года



## РАБОЧАЯ ПРОГРАММА

дисциплины «Системное программное обеспечение»  
основной образовательной программы подготовки бакалавров  
по направлению 220400- «Управление в технических системах»

Форма обучения  
Технология обучения  
Объем дисциплины

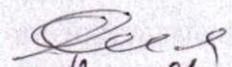
очная  
традиционная  
5 зачетных единицы

Комсомольск-на-Амуре 2011



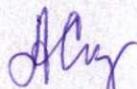
Рабочая программа обсуждена и одобрена на заседании  
кафедры «Электропривод и автоматизация промышленных установок»

Заведующий кафедрой  
д.т.н., профессор

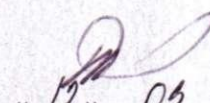
  
«12» 09 201\_ года В.А. Соловьев

## СОГЛАСОВАНО

Начальник учебно-методического  
управления к.т.н., профессор

  
«12» 09 201\_ года А.А. Скрипильев

Декан электротехнического факультета  
к.т.н., профессор

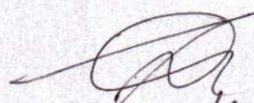
  
«12» 09 201\_ года А.Н. Степанов

Зав. кафедрой ЭПАПУ  
д.т.н., профессор


  
«12» 09 201\_ года В.А. Соловьев

Рабочая программа рассмотрена, одобрена и рекомендована к использо-  
ванию методической комиссией электротехнического факультета

Председатель методической  
комиссии к.т.н., профессор

  
«12» 09 201\_ года Н.Е. Дерюжкова

Автор рабочей программы  
к.т.н., доцент

  
«11» 09 201\_ года С.П. Черный

## Содержание

Введение.....	4
1. Пояснительная записка.....	4
1.1. Предмет, цели, задачи, принципы построения и реализации дисциплины.....	4
1.2. Роль и место дисциплины в структуре реализуемой основной образовательной программы.....	8
1.3. Характеристика трудоемкости дисциплины и ее отдельных компонентов .....	9
2. Структура и содержание дисциплины.....	11
3. Календарный график изучения дисциплины.....	11
3.1. Лекции.....	11
3.2. Лабораторные занятия.....	16
3.3. Характеристика трудоемкости, структуры и содержания самостоятельной работы студентов, график ее выполнения.....	18
4. Технологии и методическое обеспечение контроля результатов учебной деятельности обучающихся.....	20
4.1. Технологии и методическое обеспечение контроля текущей успеваемости студентов.....	20
4.2. Технологии, методическое обеспечение и условия промежуточной аттестации.....	20
4.3. Технологии, методическое обеспечение и условия отложенного контроля знаний, умений, навыков обучающихся и компетенций выпускников, сформированных в результате изучения дисциплины.....	20
5. Ресурсное обеспечение курса.....	22
5.1. Список основной учебной и учебно-методической литературы.....	22
5.2. Список дополнительной учебной и учебно-методической литературы.....	22
5.3. Перечень программных продуктов, используемых при изучении курса.....	22
Приложение А.....	23
Приложение Б.....	25
Приложение В.....	27

## **ВВЕДЕНИЕ**

Рабочая программа дисциплины «Системное программное обеспечение» разработана на основании требований Федерального государственного образовательного стандарта высшего профессионального образования по направлению подготовки 220400 «Управление в технических системах», утвержденного приказом Министерства образования и науки Российской Федерации № 813 от 22.12.2009 г.

Дисциплина «Системное программное обеспечение» является компонентом вариативной части профессионального цикла и входит в состав основной образовательной программы подготовки бакалавров по очной форме и традиционной технологии обучения. Дисциплина изучается на третьем курсе (шестой семестр).

Программа дисциплины рассчитана на студентов, получивших базовые знания по дисциплинам «Информатика», «Программные средства» и «Информационные технологии» в рамках направления 220400.62 «Управление в технических системах» утвержденного приказом Министерства образования и науки Российской Федерации № 813 от 22.12.2009 г. Содержание программы охватывает основные понятия и положения дисциплины; теорию и практику, необходимую для формирования у студентов базовых знаний, используемых в дальнейшем при изучении специальных дисциплин. Задача данного курса – освоение студентами базовых понятий проектирования системного программного обеспечения, в том числе специализированного системного программного обеспечения различного назначения и использование полученных навыков в своей профессиональной деятельности.

# 1. ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

## 1.1 Предмет, цели, задачи, принципы построения и реализации дисциплины

ФГОС ВПО по направлению подготовки 220400 «Управление в технических системах» содержит следующую характеристику

- **областей профессиональной деятельности бакалавров:**
  - проектирование, исследование, производство и эксплуатация систем и средств управления в промышленной и оборонной отраслях, в экономике, на транспорте, в сельском хозяйстве, медицине и т. п.;
  - создание современных программных и аппаратных средств исследования и проектирования, контроля, технического диагностирования и промышленных испытаний систем автоматического и автоматизированного управления;
- **объектов профессиональной деятельности бакалавров:**
  - системы автоматизации, управления, контроля, технического диагностирования и информационного обеспечения;
  - методы и средства их проектирования, моделирования, экспериментального исследования;
  - ввод в эксплуатацию на действующих объектах и техническое обслуживание этих систем;

- **профессиональных задач бакалавров, в соответствии с видами профессиональной деятельности:**

*Проектно-конструкторская деятельность:*

- участие в подготовке технико-экономического обоснования проектов создания систем и средств автоматизации и управления;
- сбор и анализ исходных данных для расчёта и проектирования устройств и систем автоматизации и управления;
- расчет и проектирование отдельных блоков и устройств систем автоматизации и управления в соответствии с техническим заданием;
- разработка проектной и рабочей документации, оформление отчетов по законченным проектно-конструкторским работам;
- контроль соответствия разрабатываемых проектов и технической документации стандартам, техническим условиям и другим нормативным документам.

*Производственно-технологическая деятельность:*

- внедрение результатов разработок в производство средств и систем автоматизации и управления;
- участие в технологической подготовке производства технических средств и программных продуктов систем автоматизации и управления;
- участие в работах по изготовлению, отладке и сдаче в эксплуатацию систем и средств автоматизации и управления;
- организация метрологического обеспечения производства;



обеспечение экологической безопасности проектируемых устройств и их производства.

*Научно-исследовательская деятельность:*

- анализ научно-технической информации, отечественного и зарубежного опыта по тематике исследования;
- участие в работах по организации и проведению экспериментов на действующих объектах по заданной методике;
- обработка результатов экспериментальных исследований с применением современных информационных технологий и технических средств;
- проведение вычислительных экспериментов с использованием стандартных программных средств с целью получения математических моделей процессов и объектов автоматизации и управления;
- подготовка данных и составление обзоров, рефератов, отчетов, научных публикаций и докладов на научных конференциях и семинарах;
- участие во внедрении результатов исследований и разработок;
- организация защиты объектов интеллектуальной собственности и результатов исследований и разработок как коммерческой тайны предприятия.

*Организационно-управленческая деятельность:*

- организация работы малых групп исполнителей;
- участие в разработке организационно-технической документации (графиков работ, инструкций, планов, смет и т.п.) и установленной отчетности по утвержденным формам;
- выполнение работ по сертификации технических средств, систем, процессов, оборудования и материалов;
- профилактика производственного травматизма, профессиональных заболеваний;
- предотвращение экологических нарушений.

*Монтажно-наладочная деятельность:*

- участие в поверке, наладке, регулировке, оценке состояния оборудования и настройке технических средств и программных комплексов автоматизации и управления на действующем объекте;
- участие в сопряжении программно-аппаратных комплексов автоматизации и управления с объектом, в проведении испытаний и сдаче в эксплуатацию опытных образцов аппаратуры и программных комплексов автоматизации и управления на действующем объекте.

*Сервисно-эксплуатационная деятельность:*

- участие в поверке, наладке, регулировке и оценке состояния оборудования и настройке аппаратно-программных средств автоматизации и управления;
- профилактический контроль технического состояния и функциональная диагностика средств и систем автоматизации и управления;

- составление инструкций по эксплуатации аппаратно-программных средств и систем автоматизации и управления и разработка программ регламентных испытаний;
- составление заявок на оборудование и комплектующие, подготовка технической документации на ремонт оборудования.

**Предметом дисциплины «Системное программное обеспечение» являются:**

- состав и свойства основных элементов системного программного обеспечения;
- элементы математической лингвистики, а также способа построения трансляторов

#### **Цели дисциплины:**

В результате освоения данной дисциплины студент приобретает знания, умения и навыки, обеспечивающие достижение целей основной образовательной программы «Управление в технических системах».

Дисциплина нацелена на подготовку студентов к:

- разработке и исследованию средств и систем проектирования системного программного обеспечения автоматизированных систем управления различного назначения, применительно к конкретным условиям производства на основе отечественных и международных нормативных документов;
- исследованию в области проектирования системного программного обеспечения и совершенствования структур и процессов при реализации вычислительных комплексов;
- исследованию с целью обеспечения высокоэффективного функционирования системного программного обеспечения средств и систем автоматизации, управления заданным.

#### **Задачи дисциплины:**

- обучение студентов теоретическим и практическим знаниям о функционировании системного программного обеспечения автоматизированных систем управления технологическими процессами, программном и информационном обеспечении АСУ ТП;
- ознакомление с современной программной реализацией различных систем и средств автоматизированных компьютерных систем, формирование навыков настройке и программированию системного программного обеспечения;
- овладение приемами и методами решения конкретных задач связанных с управлением автоматизированными компьютерными системами посредством системного программного обеспечения.

#### **Принципы построения и реализации дисциплины:**

1. Принцип соответствия требованиям ФГОС ВПО.

2. Системность и логическая последовательность представления учебного материала и его практических приложений. В разделах дисциплины вскрыты внутренние логические связи. Материал изложен так, что изучение последующего вопроса предполагает знание предыдущих.

3. Профессиональная направленность, связь теории и практики обучения с будущей профессиональной деятельностью и в целом с жизнью. Знания, умения и навыки, получаемые студентом при изучении дисциплины, будут совершенно необходимы и в последующих учебных курсах, в повседневной творческой работе, и в дальнейшем – в профессиональной деятельности.

4. От общего к частному – от общего знакомства с дисциплиной и ее теоретическими положениями и их практическими приложениями к изучению конкретных проблем с одновременной реализацией принципа «от простого к сложному». Принцип научности, обеспечивающий соответствие изучаемого материала современному состоянию и перспективам развития соответствующей области знаний, отраслей техники и технологии.

5. Принцип доступности, обеспечивающий соответствие объема и сложности учебного материала реальным возможностям студентов. Лекции составлены максимально лаконично с тем, чтобы наиболее четко и понятно изложить учебный материал. Тематика лабораторных работ закрепляет и дополняет соответствующий материал лекций, текст заданий содержит подробные указания по их выполнению.

6. Принцип опоры на практический жизненный опыт студентов. Теоретический материал иллюстрирован жизненными и понятными примерами. Лабораторные работы также содержат задания, встречающиеся и в повседневной практике, и в других дисциплинах, изучаемых студентами.

7. Принцип модульного построения дисциплины, когда каждый из компонентов-модулей дисциплины имеет определенную логическую завершенность по отношению к установленным целям и результатам обучения. Модули курса следуют друг за другом в логической последовательности. Курс дисциплины состоит из четырех модулей, которые изучаются на протяжении двух семестров.

8. Принцип формирования мотивации, положительного отношения к процессу обучения. Лекции – простые и ясные, с большим количеством интересных примеров. Лабораторные работы – увлекательные и зрелищные, с наглядными результатами. Все это формирует у студентов желание изучать дисциплину.

9. Принцип постоянного контроля, оценки и стимулирования учебных достижений обучающегося. Каждая лабораторная работа имеет контрольные вопросы, которые позволяют студентам хорошо запоминать лекционный материал не только при подготовке к экзамену, но и в процессе учебы. По выполненным лабораторным работам студенты формируют отчеты.

## **1.2 Роль и место дисциплины в структуре реализуемой основной образовательной программы**



Дисциплина «Системное программное обеспечение» относится к базовой части профессионального цикла и рекомендована ФГОС. Коррективом являются подготовка «Хранение и защита компьютерной информации», «Информационное обеспечение систем управления», «Интегрированные системы проектирования и управления автоматизированных и автоматических производств», «Информационные технологии систем управления производством».

Соответствие результатов освоения дисциплины формируемым компетенциям ООП представлено в таблице

Формируемые компетенции в соответствии с ООП	Результаты освоения дисциплины
ПК-3 ПК-15 ПК-29 ПК-31	<p><i>Знать</i></p> <p>основные понятия в системном программном обеспечении; функции и организация операционных систем; обзор современных ОС; процессы, операции над процессами; процессы и нити, идентификация и группирование процессов; классификация процессов и ресурсов, задачи синхронизации, семафорная техника синхронизации, тупики, условия возникновения, предупреждение и обходы; межпроцессорные коммуникации;</p> <p><i>Уметь:</i></p> <p>использовать планирование выполнения процессов, диспетчеризацию процессов реального времени, организацию и управление памятью, файловою систему, управление вводом/выводом, реализовывать и использовать варианты структур ядра ОС; мультипроцессорные ОС, сетевые ОС, распределенные ОС: назначение и подходы к построению;</p> <p><i>Владеть</i></p> <p>методиками реализации вычислительного процесса, обслуживания прерываний, способами управления многозадачными и многопользовательскими ОС, а также распределением ресурсов в ОС; системными программами; утилитами, макроассемблерами, компиляторами, интерпретаторами, отладчиками; сохранностью и защитой программных систем.</p>

Выпускник должен обладать следующими **профессиональными компетенциями (ПК)**:

*общепрофессиональные компетенции:*

- готовностью учитывать современные тенденции развития электроники, измерительной и вычислительной техники, информационных технологий в своей профессиональной деятельности (ПК-3);
- готовностью к участию в работах по изготовлению, отладке и сдаче в эксплуатацию систем и средств автоматизации и управления (ПК-15);
- способностью настраивать управляющие средства и комплексы и осуществлять их регламентное эксплуатационное обслуживание с использованием соответствующих инструментальных средств (ПК-29);

- готовностью производить инсталляцию и настройку системного, прикладного и инструментального программного обеспечения систем автоматизации и управления (ПК-31);

Дисциплина «Системное программное обеспечение» участвует в формировании всех вышеперечисленных знаний, умений, навыков студентов и компетенций выпускников.

Она предназначена для углубления первоначальных и формирования новых знаний, умений и навыков у учащихся по основным вопросам анализа и синтеза системного программного обеспечения.

Знания, умения и навыки, приобретенные при изучении данной дисциплины, обеспечивают успешное освоение большинства общепрофессиональных и специальных дисциплин специальности, таких как: «Автоматизированные информационно-управляющие системы», «Программные средства систем реального времени», «Локальные системы автоматизации и управления» и др.

### 1.3 Характеристика трудоемкости дисциплины и ее отдельных компонентов

**Таблица 1**

#### Характеристика трудоемкости дисциплины «Системное программное обеспечение»

Наименования показателей	Се- мест ры	Значения трудоемкости						
		Всего			в том числе:			
		зет	Часы		аудиторные занятия, часы		само- стоя- тельная работа в часах	проме- жуточ- ная ат- теста- ция (эк- замен) в часах
			всего	часов в не- делю	всего	часов в не- делю		
1. Трудоемкость дис- циплины в целом (по рабочему учебному плану программы)	-	5	180	-	72	-	72	36
2. Трудоемкость дис- циплины в семестре (по рабочему учеб- ному плану про- граммы)	6	4	144	3	72	4	72	
3. Трудоемкость по видам аудиторных занятий: - лекции	6	-	-	-	36	2	-	-
- лабораторные заня- тия	11	-	-	-	36	2	-	-
4. Промежуточная аттестация (число начисляемых зет): 4.1. Экзамены	6	1	-	-	-	-	-	36



## 2 СТРУКТУРА И СОДЕРЖАНИЕ ДИСЦИПЛИНЫ

Дисциплина условно разбита на 3 модуля:

*Модуль 1 – «Эволюция вычислительных систем»:*

примере эволюции процессоров фирмы Intel, представлены особенности в архитектуре процессоров I8080, I8086, I286, I386, вводится основополагающее понятие дескриптора, проводится анализ работы с памятью вышеперечисленных типов процессоров и реализация защищенного режима в них.

Охватываемые компетенции (ПК-3, ПК-15).

*Модуль 2 – «Формальные системы и языки программирования»:*

принципы описания и задания языков, основные виды формальных грамматик, способы построения трансляторов, компиляторов и интерпретаторов; общие сведения о языке, переводе и структуре транслятора, а также описание входного его языка; представлены связи при формировании входного языка транслятора с порождающими грамматиками и Бэкусовыми нормальными формами.

Охватываемые компетенции (ПК-15, ПК-29, ПК-31).

*Модуль 3 – «Операционная система UNIX»:*

анализ и синтез операционных систем (на примере операционной системы UNIX), в том числе архитектура, ее основные компоненты, файловая система, функции операционной системы, подсистема управления файлами, процессы, потоки, барьеры, мьютексы, семафоры, использование разделяемой памяти, проблемы межпроцессного взаимодействия, многопроцессорные системы.

Охватываемые компетенции (ПК-15, ПК-29, ПК-31).

### 3 КАЛЕНДАРНЫЙ ГРАФИК ИЗУЧЕНИЯ ДИСЦИПЛИНЫ

#### 3.1 Лекции

Таблица 2

#### Программа лекций

№ п/п	Тематика лекций	Трудоемкость (академические часы)		Ориентация материала лекции на формирование:	
		лекции в це- лом	в том числе с использова- нием актив- ных методов обучения	знаний, умений, навыков обучаю- щихся	компе- тенций выпуск- ников
1.	Эволюция процессо- ров на примере анало- гов фирмы Intel. Ар- хитектура процессо- ров I8080, I8086, I286, I386.	2		Знать эволюцию процессоров на примере аналогов фирмы Intel; ар- хитектуру про- цессоров I8080, I8086, I286, I386.	ПК-3, ПК-15
2.	Эволюция процессо- ров на примере анало- гов фирмы Intel. По- нятие дескриптора. Работа с памятью. Ре- ализация защищенно- го режима.	2	1	Знать базовые понятия операци- онных систем де- скриптор, табли- цы файлов, ин- дексов, страниц, механизмы рабо- ты с памятью, ре- ализацию защи- щенного режима	ПК-3, ПК-15
3.	Формальные системы и языки программи- рования. Принципы описания и задания языков. Формальные грамматики. Трансля- торы, компиляторы и интерпретаторы.	4		Знать формаль- ные системы и языки програм- мирования, прин- ципы описания и задания языков, формальные грамматики, трансляторы, компиляторы и интерпретаторы.	ПК-15, ПК-29, ПК-31
4.	Формальные системы и языки программи- рования. Общие све- дения о языке, пере- воде и структуре транслятора. описа- ние входного языка транслятора.	2		Знать способы формирования и описания входно- го языка трансля- тора, общие све- дения о языке, переводе и струк- туре транслятора.	ПК-15, ПК-29, ПК-31

№ п/п	Тематика лекций	Трудоемкость (академические часы)		Ориентация материала лекции на формирование:	
		лекции в це- лом	в том числе с использова- нием актив- ных методов обучения	знаний, умений, навыков обучаю- щихся	компе- тенций выпуск- ников
5.	Формальные системы и языки программирования. Бэкусовы нормальные формы. Формальные языки и грамматики. Порождающие грамматики.	4		Знать системы и языки программирования, Порождающие грамматики и Бэкусовы нормальные формы.	ПК-15, ПК-29, ПК-31
6.	Операционная система UNIX. Архитектура основные компоненты.	2		Знать основные понятия и определения, а также назначение базовых компонентов операционной системы Unix.	ПК-15, ПК-29, ПК-31
7.	Операционная система UNIX. Файловая система.	2		Знать особенности реализации и функционирования файловой системы.	ПК-15, ПК-29, ПК-31
8.	Операционная система UNIX. Функции операционной системы.	2		Знать базовые функциональные возможности операционной системы.	ПК-15, ПК-29, ПК-31
9.	Подсистема управления файлами.	2		Знать базовые понятия и определения подсистемы управления файлами.	ПК-15, ПК-29, ПК-31
10.	Процессы и потоки	4		Знать теоретические аспекты реализации и функционирования процессов и потоков в операционных системах, их основные отличия.	ПК-15, ПК-29, ПК-31



№ п/п	Тематика лекций	Трудоемкость (академические часы)		Ориентация материала лекции на формирование:	
		лекции в це- лом	в том числе с использова- нием актив- ных методов обучения	знаний, умений, навыков обучаю- щихся	компе- тенций выпуск- ников
11.	Барьеры	2		Знать базовую реализацию и особенности функционирования примитивов межпроцессорного взаимодействия.	ПК-15, ПК-29, ПК-31
12.	Использование разделяемой памяти. Семафоры. Мьютексы	4	0,5	Знать базовую реализацию и особенности функционирования примитивов межпроцессорного взаимодействия.	ПК-15, ПК-29, ПК-31
13.	Проблемы межпроцессорного взаимодействия	2		Знать основные проблемы и пути их решения при использовании различных примитивов межпроцессорного взаимодействия.	
14.	Многопроцессорные системы.	2		Знать платформы и программно-аппаратную реализацию многопроцессорных систем.	
<b>Итого в семестре 36 часов</b>					
<b>В целом по дисциплине 36 часов</b>					
<b>В том числе с использованием активных форм занятия – 1,5 часа</b>					
<sup>2)</sup> Расшифровка кодов компетенций приведена в подразделе 1.2					

### 3.2 Лабораторные занятия

Лабораторные занятия – это форма учебного занятия (под руководством преподавателя) практической работы обучающихся, направленной на закрепление и углубление, практическое подтверждение теоретических концепций курса, а также

на формирование и развитие умений и навыков планирования и проведения эксперимента, развитие умений и навыков работы с химическими методиками и химическим оборудованием.

График реализации лабораторного практикума с указанием тематики лабораторных занятий представлен в таблице 3.

**Таблица 3**

**Программа лабораторных занятий**

№ п/п	Наименования лабораторных работ	Трудоемкость (академические часы)	Основные планируемые результаты	
		занятия в целом	Знания, умения, навыки обучающихся	Компетенции выпускников
1.	Введение в ОС UNIX	6	Овладение базовыми понятиями и командами применяемыми в операционной системе UNIX.	ПК-3, ПК-15, ПК-29, ПК-31
2.	Интерпретатор SHELL	6	Уметь использовать общие положения, структуру команд, Shell-переменные, применяемые при управлении системными ресурсами средствами Shell-интерпретатора.	ПК-3, ПК-15, ПК-29, ПК-31
3.	Файловая система ОС UNIX	8	Уметь использовать принципы организации файловой системы UNIX и приобретению навыков написания программ работы с файлами.	ПК-3, ПК-15, ПК-29, ПК-31
4.	Процессы и сигналы ОС UNIX	8	Владеть навыками по созданию процессов, их управлению и синхронизации, а также способам обмена данными и сигналами между процессорами в ОС UNIX.	ПК-3, ПК-15, ПК-29, ПК-31

№ п/п	Наименования лабораторных работ	Трудоемкость (академические часы)	Основные планируемые результаты	
		занятия в целом	Знания, умения, навыки обучающихся	Компетенции выпускников
5.	Разделяемая память и семафоры в ОС UNIX	8	Ознакомится с семафорами как средством синхронизации работы параллельных процессов ОС UNIX, с обменом данными между процессами через разделяемую память; приобретение знаний и навыков написания программ работы с конкурирующими процессами.	ПК-3, ПК-15, ПК-29, ПК-31
<b>Итого в семестре: 36 часов</b>				
<b>В целом по дисциплине: 36 часов</b>				

### 3.3 Характеристика трудоемкости, структуры и содержания самостоятельной работы студентов, график ее выполнения

Объём, структура и содержание самостоятельной работы студентов, график её выполнения в 18-недельном семестре приведены в таблице 4.

Виды самостоятельной работы студентов:

- подготовка к лекциям;
- подготовка к контрольным работам;
- подготовка к лабораторным работам, оформление отчета и подготовка к защите;
- самостоятельное изучение отдельных теоретических разделов курса;
- подготовка, оформление и защита расчетно-графических заданий;
- подготовка к экзаменам по курсу.

В процессе подготовки к лекционным и лабораторным занятиям перед студентом ставится задача повторения пройденного материала, запоминания основных и ключевых понятий изучаемого предмета.

На основе результатов и защит лабораторных работ студент допускается к экзамену. Защита лабораторных работ принимается в виде индивидуальной беседы при условии оформленной лабораторной работы, решенных задач по индивидуальной карточке.



**Таблица 4**

График выполнения самостоятельной работы студентов в 18-недельном семестре

Вид самостоятельной работы	Число часов в неделю																		Итого по видам работы
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	
Подготовка к лекциям	0,4	0,3	0,3	0,4	0,3	0,3	0,3	0,4	0,3	0,3	0,3	0,4	0,3	0,3	0,3	0,4	0,3	0,4	6
Подготовка отчета по лабораторным работам и к их защите					6			6			6			6			6		30
Изучение теоретических разделов дисциплины		4		4		4		4		4		4		4		4		4	36
ИТОГО ПО КУРСУ	0,4	4,3	0,3	4,4	6,3	4,3	0,3	10,4	0,3	4,3	6,3	4,4	0,3	10,3	0,3	4,4	6,3	4,4	72

## **4 ТЕХНОЛОГИИ И МЕТОДИЧЕСКОЕ ОБЕСПЕЧЕНИЕ КОНТРОЛЯ РЕЗУЛЬТАТОВ УЧЕБНОЙ ДЕЯТЕЛЬНОСТИ ОБУЧАЕМЫХ**

### **4.1 Технологии и методическое обеспечение контроля текущей успеваемости студентов**

Для текущего контроля используется еженедельная оценка результатов учебной деятельности каждого студента с учетом, как аудиторных занятий, так и графика выполнения самостоятельной работы. Текущий контроль проводится преподавателем на лабораторных занятиях путем проверки отчетов лабораторных работ, домашнего задания, устного опроса и занимает не более 10 минут практических занятий. Учебные группы делятся на подгруппы, которые закрепляются за преподавателями. Преподаватель обеспечивает порядок проведения лабораторных работ, преемственность их с другими лабораторными работами и необходимую подготовку студентов (теоретические сведения, знание материала лекций, ознакомление с правилами техники безопасности и пожарной безопасности, качество выполнения, контрольные вопросы и графическое оформление экспериментальных данных). Методическое обеспечение лабораторного практикума представлен в приложении А.

Проведение контроля текущей успеваемости, с одной стороны, позволяет получать адекватную информацию о степени усвоения учебного материала, с другой стороны, стимулирует ритмичность учебной деятельности.

Успешно обучающимися студентами считаются те, кто к моменту промежуточной аттестации (середине семестра, концу семестра) выполнили 80% от всех заданных преподавателем на данный момент и предусмотренным учебным планом заданий.

### **4.2 Технологии и методическое обеспечение промежуточной аттестации**

Рабочим учебным планом в 6-м семестре предусмотрена промежуточная аттестация по дисциплине «Системное программное обеспечение» в форме экзамена. К аттестации допускаются студенты при наличии условий:

- выполнены и защищены в срок все лабораторные работы;
- представлены и защищены в срок расчётно-графические задания;

На экзамен выносятся теоретические вопросы в соответствии с прочитанным курсом лекций; задачи соответствуют тематике практических занятий в семестре.

Критерий оценки:

- оценка «отлично» - студент продемонстрировал знания в области построения трансляторов, правильно оперирует основными понятиями операционных систем, правильно реализует процессы, потоки и примитивы межпроцессорного взаимодействия, правильно понимает принципы построения и программно аппаратную реализацию многопроцессорных систем;

- оценка «хорошо» - студент продемонстрировал знания в области построения трансляторов, правильно оперирует основными понятиями операционных систем, но допустил некоторые ошибки при реализации процессов или потоков, не достаточно хорошо представляет принципы построения и программно аппаратную реализацию многопроцессорных систем;

- оценка «удовлетворительно» - студент продемонстрировал знания в области построения трансляторов, правильно оперирует основными понятиями операционных систем, но не определяет основные отличия при реализации процессов или потоков, не достаточно хорошо представляет принципы построения и программно аппаратную реализацию многопроцессорных систем;

Экзамен проводится по технологии совмещения письменного и устного ответов.

Дополнительные вопросы возможны только при отрицательных оценках по итогам текущей успеваемости. Экзаменационные вопросы представлены в приложении Б.

#### **4.3 Технологии, методическое обеспечение и условия отложенного контроля знаний, умений, навыков обучающихся и компетенций выпускников, сформированных в результате изучения дисциплины**

Контроль и оценка выживаемости знаний, умений и навыков, полученных при изучении дисциплины, по истечении определенного времени после аттестации, может проводиться в виде тестирования.

## **5 РЕСУРСНОЕ ОБЕСПЕЧЕНИЕ КУРСА**

### **5.1. Список основной учебной и учебно-методической литературы**

1. Керниган Б.В., Пайк Р. UNIX - универсальная среда программирования. Пер. с англ. Березко, Иващенко под ред. М.И.Белякова -М.: "Финансы и статистика", 1992.
2. Соловьев Г.Н., Никитин В.Д. Операционные системы ЭВМ. -М.: Высшая шк., 1989.
3. Т.Чан Системное программирование на C++ для UNIX. /Пер. с англ. - К.:Издательская группа BHV, 1997.
4. Волкова И.А., Руденко Т.В. “Формальные грамматики и языки. Элементы теории трансляции. (учебное пособие для студентов II курса)” - издание второе (переработанное и дополненное)- М., МГУ, 1999

### **5.2. Список дополнительной учебной, учебно-методической и научной литературы**

1. Забродин Л.Д. UNIX. Введение в командный интерфейс. -М.: "ДИАЛОГ-МИФИ",1994.-144 с.
2. Б.Ф.Мельников. Подклассы класса контекстно-свободных языков. - М., МГУ, 1995.
3. Дж.Фостер. Автоматический синтаксический анализ. - М., Мир, 1975.
4. Э. Таненбаум, «Современные операционные системы», Питер, -СПб.:, 2002г.
5. Э. Таненбаум, «Архитектура компьютеров», -СПб.: Питер, 2002 г.
6. Б. Керниган, Д. Ритчи, «Язык программирования Си», 3 изд, -СПб.: «Невский Диалект», 2001 г.
7. Дейтел Г. Введение в операционные системы: в 2-х томах. – М.: Мир, 1987.
8. Фролов А.В., Фролов Г.В. Защищенный режим процессоров Intel 80286, 80386, 80486. Практическое руководство по использованию защищенного режима. – М.: Диалог-МИФИ, 1993.
9. Э. Таненбаум, «Компьютерные сети», 3 изд. Питер, 2002г.
10. Ю.Вахалия, «Unix изнутри», -СПб.: Питер, 2003 г.

### **5.3. Другие информационные и материально-технические ресурсы**

1. [http://www.sparc.spb.su/Oio/stud/unix/unix\\_world.html](http://www.sparc.spb.su/Oio/stud/unix/unix_world.html)
2. <http://www.intuit.ru/departament/os/osunix/1/>

## **ПРИЛОЖЕНИЕ А**

Министерство образования и науки Российской Федерации

Федеральное агентство по образованию  
Государственное образовательное учреждение  
высшего профессионального образования  
«Комсомольский-на-Амуре Государственный технический универси-  
тет»  
Кафедра «Электропривод и автоматизация промышленных устано-  
вок»

### **Введение в ОС UNIX**

Методические указания к лабораторной работе по курсу  
«СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ» для студентов специ-  
альности 220201 «Управление и информатика в технических системах для  
всех форм обучения

Комсомольск-на-Амуре 2011

УДК 007.52

Введение в ОС UNIX: методические указания к лабораторной работе по курсам: «Системное программное обеспечение» /сост.: Черный С.П., Петренко Е.Д. – Комсомольск-на-Амуре : ГОУВПО «КнАГТУ», 2011. – 15 с.

В указаниях приводятся общие положения, базовые понятия и команды применяемые в операционной системе UNIX.

Предназначены для студентов специальности 220201 «Управление и информатика в технических системах для всех форм обучения.

Печатается по постановлению редакционно-издательского совета ГОУВПО «Комсомольский-на-Амуре государственный технический университет».

Согласованно с патентно-информационным отделом.

Рецензент Соловьев В.А.



### **Цель работы:**

Освоить навыки работы в операционной системе UNIX, ознакомиться с базовыми понятиями, командами и ключами операционной системы.

### **Теоретические сведения**

#### **Общие положения**

UNIX - это ядро операционной системы разделения времени, то есть программа, которая распоряжается ресурсами вычислительной машины и предоставляет их пользователям. Она дает пользователям возможность запускать свои программы, управляет периферийными устройствами и обеспечивает работу файловой системы. UNIX является многозадачной, многопользовательской ОС.

Важной составной частью UNIX является файловая система. Она имеет иерархическую структуру, образующую дерево каталогов и файлов. Корневой каталог обозначается символом "/", путь по дереву каталогов состоит из имен каталогов, разделенных символом "/", например:

`/usr/include/sys`

В каждый момент времени с любым пользователем связан текущий каталог, то есть местоположение пользователя в иерархической файловой системе.

Каждый файл ОС UNIX может быть однозначно определен некоторой структурой данных, называемой описателем файла (дескриптором). Он содержит всю информацию о файле: тип файла, режим доступа, идентификатор владельца, размер, адрес файла, даты последнего доступа и последней модификации, дату создания и пр.

Обращение к файлу происходит по имени. Локальное имя файла представляет собой набор символов, в версии System V имеющий длину от 1 до 14. В качестве символов следует использовать цифры, буквы латинского алфавита и символ '\_'. Локальное имя файла хранится в соответствующем каталоге. Путь к файлу от корневого каталога называется полным именем файла. Если обращение к файлу начинается с символа "/", то считается, что указано полное имя файла и его поиск начинается с корневого каталога, в любом другом случае поиск файла начинается с текущего каталога.

У любого файла может быть несколько имен. Фактически, имя файла является ссылкой на файл, специфицированный номером описателя.

### **Регистрация в системе**

Работа пользователя в системе начинается с того, что активизируется сервер терминального доступа `getty`, который запускает программу `login`, запрашивающую у пользователя имя и пароль.

Далее происходит проверка аутентичности пользователя в соответствии с той информацией, которая хранится в файле `/etc/passwd`. В этом файле хранятся записи, содержащие

- регистрационное имя пользователя;
- зашифрованный пароль;
- идентификатор пользователя;
- идентификатор группы;
- информация о минимальном сроке действия пароля;
- общая информация о пользователе
- начальный каталог пользователя
- регистрационный `shell` пользователя

Если пользователь зарегистрирован в системе и ввел правильный пароль, `login` запускает программу, указанную в `/etc/passwd` - регистрационный `shell` пользователя

## **Работа с файлами**

### **Пользователи системы и владельцы файлов**

Пользователь системы - это объект, обладающий определенными правами, определяющими возможность запуска программ на выполнение, а также владение файлами. Единственный пользователь системы, обладающий неограниченными правами - это суперпользователь или администратор системы.

Система идентифицирует пользователей по т.н. идентификатору пользователя (`UID` - `User Identifier`). Каждый пользователь является членом одной или нескольких групп - списка пользователей, имеющих сходные задачи. Каждая группа имеет свой уникальный идентификатор группы (`GID` - `Group Identifier`) Принадлежность группе определяет совокупность прав, которыми обладают члены данной группы.

Права пользователя UNIX - это прежде всего права на работу с файлами. Файлы имеют двух владельцев - пользователя (`user owner`) и группу (`group owner`).

Соответственно атрибуты защиты файлов определяют права пользователя-владельца файла (`u`), права члена группы-владельца (`g`) и права всех остальных (`o`).

## **Перенаправление потоков и программные каналы**

В ОС UNIX существует три стандартных потока: поток ввода, поток вывода и поток стандартного протокола (поток ошибок).

Перенаправление потоков позволяет изменить стандартный ввод/вывод:

< - изменение источника стандартного ввода;

>, >> - изменение приемника стандартного вывода.

Примеры:

**cat > filename** - перенаправление вывода программы cat в файл filename (если этот файл существует, то его прежнее содержимое будет утеряно);

**cat >> filename** - добавить содержимое вывода программы cat к содержимому файла filename;

**cat < filename** - сформировать стандартный ввод программы cat из содержимого файла filename.

Стандартные потоки - поток ввода, поток вывода и поток ошибок (поток протокола) имеют фиксированную нумерацию - 0, 1 и 2 соответственно. Эти номера (номера дескрипторов потоков) можно использовать в явном виде. Например, запись

prog 1>file

эквивалентна записи

prog >file

Для того, чтобы отличить имя потока от имени файла, перед номером потока ставится символ '&':

prog >file 2>&1

Здесь происходит перенаправление стандартного потока вывода в файл file (>file). А кроме того, сообщения об ошибках также будут перенаправлены в файл file: запись 2>&1 означает перенаправление потока ошибок на стандартный поток вывода, который, в свою очередь, был перенаправлен в файл.

Замечание: анализ команды осуществляется интерпретатором справа налево: сначала происходит слияние потоков (2>&1), а затем перенаправляется стандартный поток вывода (1) в файл file.

В связи с этим бывает полезно использование псевдоустройства /dev/null, удаляющего все введенные в него символы. Это используется тогда, когда необходимо полностью игнорировать (подавить) выходные потоки.

**Канал** - это программное средство, связывающее процессы ОС UNIX буфером ввода/вывода. Запуск процессов в виде

**\$ процесс\_1 | процесс\_2 | ... | процесс\_n**

означает, что стандартный вывод процесса\_1 будет замкнут на стандартный ввод процесса\_2, стандартный вывод процесса\_2 будет замкнут на стандартный ввод процесса\_3 и т.д. При этом сначала создается канал, а

потом на выполнение одновременно запускаются все процессы, и общее время их выполнения определяется более медленным процессом.

Пример: `ls | wc -l`

Те же действия можно организовать так:

`ls > buffer`

`wc -l < buffer`

`rm -f buffer`

Команда `ls` выводит на экран (стандартный поток вывода) список файлов текущего каталога, а команда `wc -l` считает количество строк во входном потоке (в файле, а если файл не указан - в стандартном входном потоке). Таким образом, объединение этих двух команд программным каналом позволяет посчитать количество файлов в текущем каталоге.

Итоговая таблица:

<code>&gt; file</code>	Перенаправление стандартного потока вывода в файл <code>file</code>
<code>&gt;&gt; file</code>	Добавление в файл <code>file</code> данных из стандартного потока вывода
<code>&lt; file</code>	Получение стандартного потока ввода из файла <code>file</code>
<code>p1   p2</code>	Передача стандартного потока вывода программы <code>p1</code> в поток ввода программы <code>p2</code>
<code>n &gt; file</code>	Переключение потока вывода из файла с дескриптором <code>n</code> в файл <code>file</code>
<code>n &gt;&gt; file</code>	Добавление записей потока вывода из файла с дескриптором <code>n</code> в файл <code>file</code>
<code>n &gt; &amp;m</code>	Слияние потоков с дескрипторами <code>n</code> и <code>&amp;m</code>

### Командный язык системы Unix

Набор имени команды производится с клавиатуры после появления промпта (приглашения), обычно, - \$.

Для облегчения работы с системой UNIX имеется возможность использовать шаблоны имен файлов (или метасимволы):

? - один любой символ;

\* - произвольное количество любых символов.

Например:

\*.c - задает все файлы с расширением "c";

pr????\* - задает файлы, имена которых начинаются с "pr", содержат пять символов и имеют любое расширение.

### Справочные команды

**date** - получение даты и времени.

**who** - получение списка пользователей, работающих в системе в данный момент.

Пример:

```
$ who
```

```
petr tty4i Mar 11 18:46
```

```
ann tty12 Mar 11 16:29
```

Выводится имя пользователя, номер терминала, дата и время начала работы этого пользователя. Команда `who am i` выведет информацию о самом пользователе.

**man** - получение справочной информации:

```
$ man [имя команды]
```

Альтернативой команды `man` в некоторых клонах UNIX является команда `use`:

```
use [имя команды]
```

**Команды работы с каталогами**

**pwd** - печать имени текущего каталога.

Например:

```
$ pwd
```

```
/usr/work/petr
```

**ls** - вывод на экран содержимого каталога:

```
$ ls [-ключи] [имя каталога]
```

Если имя каталога не указано, выводится содержимое текущего каталога. Ключи определяют формат выдачи, например:

-l - вывод полной информации о каждом файле;

-a - вывод полного списка файлов, включая "." и "..";

-t - сортировка списка по времени создания;

-C - вывод списка в несколько колонок по алфавиту и т.п.

Пример:

```
$ ls -l
```

```
total 5
```

```
drwxrwxrwx  6      petr  user1   496    Mar 10 12:01    tmp
```

-rw-rw-r--	1	petr	user1	156	Mar 12 15:26	file.c
-rwxrwx--x	2	root	root	4003	Apr 01 11:44	pe.out
права до-	Число	имя	имя	длина	дата послед-	локальное
ступа	ссылки	влад.	группы	файла,	ней модифи	имя файла
			влад.	байт	кации	

**cd** - смена директории (каталога):

**\$ cd [полное\_имя\_каталога]**

При этом указанный каталог станет текущим. Команда **cd** без аргументов восстановит в качестве текущего каталога начальный каталог пользователя.

**mkdir** - создание нового каталога:

**\$ mkdir [-ключи] имя\_нового\_каталога**

Для создания нового каталога пользователь должен иметь право записи в родительский каталог текущего каталога.

**rmdir** - удаление директории:

**\$ rmdir список\_каталогов**

Система не позволит удалить каталог, если он не пуст или если у пользователя нет прав записи в него. Текущий каталог не должен принадлежать поддереву удаляемых каталогов.

### Команды работы с файлами

**rm** - удаление файлов (ссылки на файл):

**\$ rm [-ключи] список\_файлов**

Эта команда удаляет ссылки на файлы (то есть локальные имена файлов), если у пользователя есть право записи в каталог, содержащий эти имена. Если удаляемый файл защищен от записи, команда запрашивает подтверждение на удаление файла.

Ключи:

-i - вводит необходимость подтверждения для каждого удаляемого файла;

-f - отменяет необходимость подтверждения для любого удаляемого файла;

-r - задает режим рекурсивного удаления всех файлов и подкаталогов данного каталога, а затем и самого каталога.

**chmod** - изменение атрибутов защиты файла:

**\$ chmod атрибуты список\_файлов**

Атрибуты файла могут быть заданы разными способами:

1) буквенной кодировкой. Атрибуты защиты обозначаются так:

r - доступ по чтению;



w - доступ по записи;

x - доступ по исполнению.

Категории пользователей задаются следующим образом:

u - атрибуты для владельца файла;

g - атрибуты для группы владельца файла;

o - атрибуты для прочих пользователей;

a - атрибуты для всех категорий пользователей.

Производимая операция кодируется с помощью таких символов:

= - установить значения всех атрибутов для данной категории пользователей;

+ - добавить атрибут для данной категории пользователей;

- - исключить атрибут для данной категории пользователей.

Пример. Разрешить доступ по чтению и записи к файлам с именем `mar` владельцу и группе-владельцу:

```
$ chmod ug+rw mar.*
```

2) в виде восьмеричного числа. Числовые значения атрибутов защиты кодируются трехразрядным восьмеричным числом, где существование соответствующего атрибута соответствует наличию единицы в двоичном эквиваленте восьмеричной цифры этого числа, а отсутствие атрибута - нулю. Например:

символьное представление	gwx	r-x	r--
двоичное представление	111	101	100
восьмеричное представление 7	5	4	

Пример. Установить атрибуты чтения и записи для владельца и группы-владельца и только чтения для остальных пользователей:

```
$ chmod 0664 gb??.doc
```

**cat** - слияние и вывод файлов на стандартное устройство вывода:

```
$ cat [-ключи] [входной_файл1[входной_файл2...]]
```

Команда по очереди читает указанные входные файлы, если их несколько, объединяет и выводит считанные данные в стандартный поток вывода (на экран). С помощью перенаправления потоков (программных каналов) команда `cat` может быть использована для выполнения разнообразных операций.

Примеры:

1) `$ cat > file1`

- в файл `file1` помещается текст, набираемый на клавиатуре.

Если до этого файл file1 не существовал, он будет создан; если существовал, его первоначальное содержимое будет утрачено. Окончание ввода текста происходит при нажатии комбинации клавиш Ctrl+D.

2) `$ cat file1 > file2`

- содержимое файла file1 копируется в файл file2. Файл file1 при этом остается без изменений.

3) `$ cat file1 file2 > result`

- содержимое file2 будет добавлено к содержимому file1 и помещено в файл result.

4) `$ cat file1 >> file2`

- содержимое файла file1 добавляется в конец файла file2.

**cp** - копирование файлов:

**`$ cp vx_файл_1 [vx_файл_2 [...vx_файл_n]] вых_файл`**

Эта команда имеет два режима использования:

1. если выходной файл есть обычный файл, то входной файл может быть только один; его содержимое копируется в выходной файл. Если выходной файл существовал, то его старое содержимое утрачивается, а атрибуты защиты остаются; если выходной файл не существовал, то он будет создан и унаследует атрибуты входного файла.

2. если выходной файл есть каталог, то в него скопируются все указанные входные файлы, но каталог естественно должен быть создан заранее.

Пример. Скопировать два файла из текущего каталога в указанный с теми же именами:

```
$ cp f1.txt f2.txt ../usr/petr
```

```
$ ls ../usr/petr
```

```
f1.txt
```

```
f2.txt
```

**mv** - пересылка файлов:

**`$ mv vx_файл_1 [vx_файл_2 [...vx_файл_n]] вых_файл`**

Отличие команды пересылки от команды копирования состоит только в том, что входные файлы после выполнения команды уничтожаются.

Пример. Перенести файлы с расширением "c" из указанного каталога в текущий:

```
$ mv petr/*.c .
```

**ln** - создание новых ссылок на файл:

**\$ ln vx\_файл\_1 [vx\_файл\_2 [...vx\_файл\_n]] вых\_файл**

Эта команда имеет два режима использования:

1. Если выходной файл есть обычный файл, то входной файл может быть только один; в этом случае на него создается ссылка с именем вых\_файл и к нему можно обращаться и по имени vx\_файл\_1, и по имени вых\_файл. Количество ссылок на файл в описателе увеличивается на 1.
2. Если выходной файл есть каталог, то в нем создаются элементы, включающие имена всех перечисленных входных файлов и ссылки на них.

Пример:

```
$ ls
file1
$ ln file1 file2
$ ls
file1
file2
```

### **Команды работы с текстовыми файлами**

**grep** - поиск шаблона (подстроки) в файлах:

**\$ grep [-ключи] подстрока список\_файлов**

Найденные строки выводятся на стандартный вывод в формате, определяемом ключами. Если файлов несколько, то перед каждой строкой выводится имя соответствующего файла. Ключи:

- с - вывод имен всех файлов с указанием количества строк, содержащих шаблон;
- i - игнорирование регистра (различия строчных и заглавных латинских букв);
- n - вывод перед строкой ее относительного номера в файле;
- v - вывод строк, не содержащих шаблона (инверсия вывода);
- l - вывод только имен файлов, содержащих шаблон.

**wc** - подсчет количества строк, слов и символов в файлах:

**\$ wc [-lwc] [список\_файлов]**

Подсчет строк - ключ -l, слов - ключ -w и символов - ключ -c (по умолчанию -lwc). Если список файлов пуст, то подсчет ведется в стандартном потоке ввода.

**sort** - сортировка файлов:

**\$ sort [-ключи] список\_файлов**

Эта команда сортирует входные файлы по строкам в соответствии с увеличением кодов символов. Ключи:

- r - обратный порядок сортировки;
- f - не учитывать различие строчных и прописных латинских букв
- n - числовой порядок сортировки и т.д.

**cmp** - вывод места первого расхождения:

**\$ cmp файл\_1 файл\_2**

Выводит номер символа и номер строки (в текстовых файлах), в которой впервые встречается расхождение во входных файлах. Работает с любыми файлами.

**diff** - вывод всех расхождений в файлах:

**\$ diff файл\_1 файл\_2**

Выводит все строки, в которых встречаются расхождения между входными файлами. Работает только с текстовыми файлами.

**find** - поиск файлов в поддереве каталогов:

**find список\_каталогов условия\_поиска**

Команда последовательно просматривает все поддеревья, начинающиеся с одного из каталогов, указанных в списке каталогов, анализирует их атрибуты, и если они удовлетворяют условиям поиска: выполняет действия, заданные в условиях\_поиска

В команде может быть задано множество условий поиска, необходимые комбинации которых объединяются в булевское выражение с помощью логических операций:

- |                 |                               |
|-----------------|-------------------------------|
| ! условие       | отрицание условия;            |
| пробел          | соответствует операции «И»;   |
| -o              | операция «ИЛИ»;               |
| \( выражение \) | булевское выражение в скобках |

Перечислим некоторые опции, задающие условия (при этом условимся обозначать через n положительное десятичное число, +n - любое положительное число, строго большее n, -n - любое положительное число, строго меньшее n):

- **-name имя\_файла** истинно для файлов с именем имя\_файла; в задаваемом имени допускается использование метасимволов;
- **-perm 8-ричный\_код** истинно для файлов с указанным кодом прав доступа;
- **-type {f|d|b|c|p}** истинно для файлов указанного типа (f - обычный файл, d - каталог, b - блок-ориентированный специальный файл, c - байт-ориентированный специальный файл, p - именованный канал);
- **-print** всегда истинно; вызывает печать маршрутного имени файла;

- **-size n[c]** истинно для файлов с длиной n. По умолчанию длина задается в блоках по 512 байт, а если после длины ставится символ c, то в байтах;
- **-exec команда** истинно, если команда, выполняющаяся при наличии данного условия, возвращает нулевой код завершения. Если в тексте команды должно быть указано имя текущего проверяемого файла, то вместо него пишут { }. В конце команды должна стоять экранированная точка с запятой: '\;';
- **-links n** истинно для файлов с числом ссылок n.

### **Команды работы с процессами**

**&** - запуск процесса как фонового (параллельного):

\$ имя\_процесса [-ключи] [параметры] &

При выполнении этой команды следующее приглашение ОС появится сразу же после запуска процесса (не дожидаясь его завершения). Фоновый процесс не допускает ввода с клавиатуры и выводит сообщения на экран, нарушая целостность ввода и вывода привилегированного процесса.

**nohup** - корректный запуск процесса как фонового:

\$ nohup имя\_процесса [-ключи] [параметры]

Эта команда перенаправляет поток вывода фонового процесса в файл nohup.out.

**ps** - получить список всех процессов:

\$ ps [-ключи]

При отсутствии ключей будет выведен список процессов самого пользователя (идентификатор процесса, номер терминала и время процессора, затраченное на процесс). Ключи:

-e - вывод информации обо всех процессах в системе;

-a - вывод информации о процессах, связанных с данным терминалом;

-l - вывод информации в длинном формате.

**kill** - послать сигнал процессу:

\$ kill -номер\_сигнала идентификатор\_процесса

Для принудительного завершения процесса ему посылается сигнал номер 9, который невозможно проигнорировать или обработать в процессе никаким иным образом, кроме немедленного завершения.

### **Задания на лабораторную работу**

Перед началом выполнения работы необходимо получить у преподавателя имя (идентификатор пользователя) и пароль.

При запуске система запрашивает login (идентификатор пользователя) и password (пароль). При успешном вводе появляется приглашение (обычно - \$); в противном случае необходимо еще раз повторить ввод. (Пароль при вводе на экране не отображается).

1. Посчитать количество пользователей в системе.
2. Отсортировать список файлов текущей директории в обратном порядке и записать его в файл.
3. Посчитать количество файлов текущего каталога, содержащих подстроку "include".
4. Посчитать, сколько раз пользователь X вошел в систему.
5. Отсортировать список текстовых файлов текущей директории в алфавитном порядке и записать его в файл.
6. Удалить из текущего каталога все файлы, содержащие подстроку "text".
7. Объединить все файлы с расширением ".txt" в один файл.
8. Посчитать, сколько процессов запущено с данного терминала.
9. Вывести на экран отсортированный в алфавитном порядке список файлов, содержащих подстроку "include".



## Литература

1. Дансмур М., Дейвис Г. Операционная система UNIX и программирование на языке Си: Пер. с англ.- М.: "Радио и связь", 1989.-192 с.
2. Забродин Л.Д. UNIX. Введение в командный интерфейс. -М.: "ДИАЛОГ-МИФИ", 1994.-144 с.
3. Керниган Б.В., Пайк Р. UNIX - универсальная среда программирования: Пер. с англ.- М.: Финансы и статистика, 1992.-304 с.
4. Робачевский А.М. Операционная система UNIX. - СПб.: BHV - Санкт-Петербург, 1997. - 528 с.
5. Т. Чан Системное программирование на C++ для UNIX. /Пер. с англ. -К.: Издательская группа BHV, 1997. - 592 с.

Министерство образования и науки Российской Федерации

Федеральное агентство по образованию  
Государственное образовательное учреждение  
высшего профессионального образования  
«Комсомольский-на-Амуре Государственный технический универси-  
тет»  
Кафедра «Электропривод и автоматизация промышленных устано-  
вок»

### **Интерпретатор SHELL**

Методические указания к лабораторной работе по курсу  
«СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ» для студентов специ-  
альности 220201 «Управление и информатика в технических системах» для  
всех форм обучения

Комсомольск-на-Амуре 2011

УДК 007.52

Интерпретатор Shell: методические указания к лабораторной работе по курсам: «Системное программное обеспечение» /сост.: Черный С.П., Петренко Е.Д. – Комсомольск-на-Амуре : ГОУВПО «КнАГТУ», 2011. – 10 с.

В указаниях приводятся общие положения, структура команд, Shell-переменные, применяемые при управлении системными ресурсами средствами Shell-интерпретатора.

Предназначены для студентов специальности 220201 «Управление и информатика в технических системах для всех форм обучения.

Печатается по постановлению редакционно-издательского совета ГОУВПО «Комсомольский-на-Амуре государственный технический университет».

Согласованно с патентно-информационным отделом.

Рецензент Соловьев В.А.

### **Цель работы:**

Изучение основных возможностей программирования на уровне командного языка Shell путём написания Shell-программ для работы с файловой системой.

### **Теоретические сведения**

#### **Общие положения**

Интерпретатор SHELL является оболочкой над всей операционной системой и выполняет интерфейсные функции между пользователем и ОС. Он перехватывает и интерпретирует все команды пользователя: формирует и выводит ответные сообщения.

Помимо запуска на выполнение стандартных команд UNIX и исполняемых файлов, интерпретатор включает собственный язык, который по своим возможностям приближается к высокоуровневым языкам программирования. Этот язык позволяет создавать программы (shell-файлы), которые могут включать операторы языка и команды UNIX. Такие файлы не требуют компиляции и выполняются в режиме интерпретации, но они должны обладать правом на исполнение (устанавливается с помощью команды `chmod`).

Процедуре shell могут быть переданы аргументы при запуске. Каждому из первых девяти аргументов ставится в соответствие позиционный параметр от \$1 до \$9 (\$0 - имя самой процедуры), и по этим именам к ним можно обращаться из текста процедуры.

Прежде, чем начать рассмотрение некоторых операторов shell, следует обратить внимание на использование в командах некоторых символов.

\        знак отмены специального символа перевода строки, следующего непосредственно вслед за этим знаком.

“”        одинарные кавычки: используются для обрамления текста: передаваемого как единый аргумент команды.

“”        двойные кавычки, используются для обрамления текста, содержащего имена переменных для подстановки (\$имя) или стандартные команды, заключенные в символы тупого ударения (`команда`).

` `        символы тупого ударения, служат для выполнения команды, заключенной между ними.

## Структура команд

Команды в Shell имеют следующий формат:

<имя команды><флаги><аргументы>

**В таблице 1 представлены некоторые средства группировки команд, которые могут быть использованы при создании командных файлов на shell.**

**Таблица 1 – Средства группировки команд**

Средства группировки	Пояснение
;	определяет последовательное выполнение команд
&	определяет асинхронное (фоновое) выполнение команд
&&	определяет выполнение последующей команды при нормальном завершении предыдущей
	определяет выполнение последующей команды при ненормальном завершении предыдущей

Например,

`k1&& k2; k3`

`k2` будет выполнена при успешном выполнении `k1`; `k3` будет выполнена после любого из исходов обработки `k2`

`k1&&{k2;k3}` - `k2`, `k3` будут выполнены при успешном выполнении `k1`

`{k1;k2}&` - в фоновой режиме будет выполняться последовательность команд `k1`, `k2`

## Синхронное и асинхронное выполнение команд

Обычно shell ждет завершения выполнения команды. Однако имеется возможность запустить задачу в асинхронном режиме, т.е. без ожидания ее завершения. Для этого после команды (после всех ее аргументов и указаний о переназначении ввода-вывода) надо поставить знак `&`. При этом по умолчанию стандартный ввод команды назначается на пустой файл `/dev/null`.

Пример: создать файл `primer` можно по команде  
`echo > primer`

Еще пример: запустить программу `prog` в асинхронном режиме, чтобы не надо было дожидаться его завершения, засечь время выполнения, результаты программы направить в файл `prog.res`, данные о времени выполнения - в файл `prog.tim`.

`time prog > prog.res 2> prog.tim &`

**echo** вывод сообщений.

*Пример.* Вывод на экран содержимого текущего каталога.

```
echo "Текущий каталог: \  
`pwd` \  
`ls`"
```

Будет выведено:

```
Текущий каталог: имя_каталога  
файл_1  
файл_2
```

## Переменные языка shell

Язык shell позволяет работать с переменными (без предварительного объявления). Имена переменных начинаются с буквы и могут включать буквы, цифры и знака подчеркивания (\_). Обращение к переменным начинается со знака \$.

Использование значения переменной называется подстановкой.

Различается два класса переменных: позиционные и с именем. Позиционные переменные - это аргументы командных файлов, их именами служат цифры: \$0 - имя команды, \$1 - первый аргумент и т.д. Значения позиционным переменным могут быть присвоены и командой set (см. Специальные команды).

**Пример.** После вызова программы на shelle, хранящейся в файле `ficofl`:

```
ficofl -d / *.for
```

значением \$0 будет `ficofl`, \$1 - `-d`, \$2 - `/`, \$3 - `*.for`, значения остальных позиционных переменных будут пустыми строками. Заметим, что если бы символ \* при вызове `ficofl` не был экранирован, в качестве аргументов передались бы имена всех фортранных файлов текущей директории.

Еще две переменные хранят командную строку за исключением имени команды: \$@ эквивалентно \$1 \$2 ..., а \$\* - "\$1 \$2 ...". Начальные значения переменным с именем могут быть установлены следующим образом:

```
<имя>=<значение> [ <имя>=<значение> ]
```

Не может быть одновременно функции (см. Управляющие конструкции) и переменной с одинаковыми именами. Для подстановки значений переменных возможны также следующие конструкции:

```
${<переменная>}
```

если значение <переменной> определено, то оно подставляется.

Скобки применяются лишь если за <переменной> следует символ, который без скобок приклеится к имени.



**`${<переменная>:-<слово>}`**

если `<переменная>` определена и не является пустой строкой, то подставляется ее значение; иначе подставляется `<слово>`.

**`${<переменная>:=<слово>}`**

если `<переменная>` не определена или является пустой строкой, ей присваивается значение `<слово>`; после этого подставляется ее значение.

**`${<переменная>:?<слово>}`**

если `<переменная>` определена и не является пустой строкой, то подставляется ее значение; иначе на стандартный вывод выводится `<слово>` и выполнение `shella` завершается. Если `<слово>` опущено, то выдается сообщение "parameter null or not set".

**`${<переменная>:+<слово>}`**

если `<переменная>` определена и не является пустой строкой, то подставляется `<слово>`; иначе подставляется пустая строка.

**Пример:** если переменная `d` не определена или является пустой строкой, то выполняется команда `pwd`

```
echo ${d:-`pwd`}
```

Следующие переменные автоматически устанавливаются `shell`'ом:

- # количество позиционных параметров (десятичное)
- флаги, указанные при запуске `shella` или командой `set`
- ? десятичное значение, возвращенное предыдущей синхронно выполненной командой
- \$ номер текущего процесса
- ! номер последнего асинхронного процесса
- @ эквивалентно `$1 $2 $3 ...`
- \* эквивалентно `"$1 $2 $3 ..."`

Напомним: чтобы получить значения этих переменных, перед ними нужно поставить знак `$`. Пример: выдать номер текущего процесса:

```
echo $$
```

## Оператор присваивания

Присвоение значений переменным осуществляется с помощью оператора `'='` без пробелов.

Пример. `s=Hello`

```
echo $s
```

## Вычисление выражений

Команда **expr** применяется для вычисления выражений. Результат выводится на стандартный вывод. Операнды выражения должны быть разделены пробелами. Метасимволы должны быть экранированы. Надо заметить, что 0 возвращается в качестве числа, а не для индикации пустой строки. Строки, содержащие пробелы или другие специальные символы, должны быть заключены в кавычки. Целые рассматриваются как 32-битные числа.

Осуществляется с помощью команды **expr** и операторов:

	арифметических		логических
+	сложение	=	равно
-	вычитание	!=	не равно
\*	умножение	\<	меньше
/	деление	\<=	меньше или равно
%	остаток от деления	\>	больше
		\>=	больше или равно

Результат операций сравнения - вывод 1 (true) или 0 (false). Все операторы и имена переменных должны отделяться друг от друга пробелами.

*Пример.*

```
a=5 b=12
a=`expr $a + 4`
d=`expr $b - $a`
echo $a $b $d
```

Будет выведено:

```
9 12 3
```

## Специальные переменные

Shell'ом используются следующие специальные переменные:

**HOME** директория, в которую пользователь попадает при входе в систему или при выполнении команды **cd** без аргументов

**PATH** список полных имен каталогов, в которых ищется файл при указании его неполного имени.

**PS1** основная строка приглашения (по умолчанию \$)

**PS2** дополнительная строка приглашения (по умолчанию >); в интерактивном режиме перед вводом команды shell'ом выводится основная строка приглашения.

Если нажата клавиша **new\_line**, но для завершения команды требуется дальнейший ввод, то выводится дополнительная строка приглашения

**IFS** последовательность символов, являющихся разделителями в командной строке (по умолчанию это <пробел>, <табуляция> и <возврат\_каретки>)

## Команда **ena**

Команда **ena** позволяет получить части полного имени файла. Первый аргумент - флаг, второй - имя файла. Команда различает следующие флаги:

- n     - имя файла без расширения
- f     - имя файла с расширением
- e     - расширение
- d     - имя директории
- p     - если имя файла начинается с . или .. , то эти символы выделяются из имени

## Функции

```
<имя> () {
    <список>;
}
```

Определяется функция с именем <имя>. Тело функции - <список>, заключенный между { и }.

## Специальные команды

Как правило, для выполнения каждой команды shell порождает отдельный процесс. Специальные команды отличаются тем, что они встроены в shell и выполняются в рамках текущего процесса.

**:**       Пустая команда. Возвращает нулевой код завершения.

**file**   Shell читает и выполняет команды из файла file, затем завершается; при поиске file используется список поиска \$PATH.

**break [n]**   Выход из внутреннего for или while цикла; если указано n, то выход из n внутренних циклов.

**continue [n]**   Перейти к следующей итерации внутреннего for или while цикла; если указано n, то переход к следующей итерации n-ого цикла.

**cd [ <аргумент> ]**   Сменить текущую директорию на директорию <аргумент>. По умолчанию используется значение HOME.

**echo [ <арг> ... ]**   Выводит свои аргументы в стандартный вывод, разделяя их пробелами.

**eval [ <арг> ... ]**   Аргументы читаются, как если бы они поступали из стандартного ввода и рассматриваются как команды, которые тут же и выполняются.

**exec [ <arg> ... ]** Аргументы рассматриваются как команды shell'a и тут же выполняются, но при этом не создается нового процесса. В качестве аргументов могут быть указаны направления ввода-вывода и, если нет никаких других аргументов, то будет изменено лишь направление ввода-вывода текущей программы.

**exit [ n ]** Завершение выполнения shell'a с кодом завершения n. Если n опущено, то кодом завершения будет код завершения последней выполненной команды (конец файла также приводит к завершению выполнения).

**export [ <переменная> ... ]** Данные переменные отмечаются для автоматического экспорта в окружение (см. Окружение) выполняемых команд. Если аргументы не указаны, то выводится список всех экспортируемых переменных. Имена функций не могут экспортироваться.

**hash [ -r ] [ <команда> ... ]** Для каждой из указанных команд определяется и запоминается путь поиска. Опция -r удаляет все запомненные данные. Если не указан ни один аргумент, то выводится информация о запомненных командах: hits - количество обращений shell'a к данной команде; cost - объем работы для обнаружения команды в списке поиска; command - полное имя команды. В некоторых ситуациях происходит перерасчет запомненных данных, что отмечается значком \* в поле hits.

**pwd** Выводит имя текущей директории.

**read [ <переменная> ... ]** Читается из стандартного ввода одна строка; первое ее слово присваивается первой переменной, второе - второй и т.д., причем все оставшиеся слова присваиваются последней переменной.

**readonly [ <переменная> ... ]** Запрещается изменение значений указанных переменных. Если аргумент не указан, то выводится информация обо всех переменных типа readonly.

**return [ n ]** Выход из функции с кодом завершения n. Если n опущено, то кодом завершения будет код завершения последней выполненной команды.

**set [ --aefkntuvx [ <arg> ... ] ]** Команда устанавливает следующие режимы:

- a отметить переменные, которые были изменены или созданы, как переменные окружения (см. Окружение)
- e если код завершения команды ненулевой, то немедленно завершить выполнение shell'a
- f запретить генерацию имен файлов
- k все переменные с именем помещаются в окружение команды, а не только те, что предшествуют имени команды (см. Окружение)
- n читать команды, но не выполнять их
- t завершение shell'a после ввода и выполнения одной команды

-u     при подстановке рассматривать неустановленные переменные как ошибки  
 -v     вывести вводимые строки сразу после их ввода  
 -x     вывести команды и их аргументы перед их выполнением  
 --     не изменяет флаги, полезен для присваивания позиционным переменным новых значений.

При указании + вместо - каждый из флагов устанавливает противоположный режим. Набор текущих флагов есть значение переменной \$- . <arg> - это значения, которые будут присвоены позиционным переменным \$1, \$2 и т.д. Если все аргументы опущены, выводятся значения всех переменных.

**shift [ n ]**     Позиционные переменные, начиная с \$(n+1), переименовываются в \$1 и т.д. По умолчанию n=1.

**test**     вычисляет условные выражения (см. Дополнительные сведения. Test )

**trap [ <arg> ] [ n ] ...**     Команда <arg> будет выполнена, когда shell получит сигнал n (см. Сигналы). (Надо заметить, что <arg> проверяется при установке прерывания и при получении сигнала). Команды выполняются по порядку номеров сигналов. Любая попытка установить сигнал, игнорируемый данным процессом, не обрабатывается. Попытка прерывания по сигналу 11 (segmentation violation) приводит к ошибке. Если <arg> опущен, то все прерывания устанавливаются в их начальные значения. Если <arg> есть пустая строка, то этот сигнал игнорируется shell'ом и вызывается им программами. Если n=0, то <arg> выполняется при выходе из shell'a. Trap без аргументов выводит список команд, связанных с каждым сигналом.

**type [ <имя> ... ]**     Для каждого имени показывает, как оно будет интерпретироваться при использовании в качестве имени команды: как внутренняя команда shell'a, как имя файла или же такого файла нет вообще.

**ulimit [ -f ] [ n ]**     Устанавливает размер файла в n блоков; -f - устанавливает размер файла, который может быть записан процессом-потомком (читать можно любые файлы). Без аргументов - выводит текущий размер.

**umask [ nnn ]**     Пользовательская маска создания файлов изменяется на nnn. Если nnn опущено, то выводится текущее значение маски. Пример: после команды umask 755 будут создаваться файлы, которые владелец сможет читать, писать и выполнять, а все остальные - только читать и выполнять.

**unset [ <имя> ... ]**     Для каждого имени удаляет соответствующую переменную или функцию. Переменные PATH, PS1, PS2 и IFS не могут быть удалены.

**wait [ n ]** Ждет завершения указанного процесса и выводит код его завершения. Если n не указано, то ожидается завершения всех активных процессов-потомков и возвращается код завершения 0.

## Выполнение shell-программ

### Запуск shell'a

Программа, интерпретирующая shell-программы, находится в файле /bin/sh. При запуске ее первый аргумент является именем shell-программы, остальные передаются как позиционные параметры. Если файл, содержащий shell-программу, имеет право выполнения (x), то достаточно указания лишь его имени. Например, следующие две команды операционной системы эквивалентны (если файл `ficofl` обладает указанным правом и на самом деле содержит shell-программу):

```
sh ficofl -d . g\*
```

и

```
ficofl -d . g\*
```

### Выполнение

При выполнении shell-программ выполняются все подстановки. Если имя команды совпадает с именем специальной команды, то она выполняется в рамках текущего процесса. Так же выполняются и определенные пользователем функции. Если имя команды не совпадает ни с именем специальной команды, ни с именем функции, то порождается новый процесс и осуществляется попытка выполнить указанную команду.

Переменная `PATH` определяет путь поиска директории, содержащей данную команду. По умолчанию это

```
::/bin:/usr/ bin:/util:/dss/rk
```

Директории поиска разделяются двоеточиями; `::` означает текущую директорию. Если имя команды содержит символ `/`, значение `$PATH` не используется: имена, начинающиеся с `/` ищутся от корня, остальные - от текущей директории. Положение найденной команды запоминается `shell`ом и может быть опрошено командой `hash`.

### Окружение

Окружение - это набор пар имя-значение, которые передаются выполняемой программе. Shell взаимодействует с окружением несколькими

способами. При запуске shell создает переменную для каждой указанной пары, придавая ей соответствующее значение. Если вы измените значение какой-либо из этих переменных или создадите новую переменную, то это не окажет никакого влияния на окружение, если не будет использована команда `export` для связи переменной shell'a с окружением (см. также `set -a`). Переменная может быть удалена из окружения командой `unset` (см.). Таким образом, окружение каждой из выполняемых shell'ом команд формируется из всех неизмененных пар имя-значение, первоначально полученных shell'ом, минус пары, удаленные командой `unset`, плюс все модифицированные и измененные пары, которые для этого должны быть указаны в команде `export`.

Окружение простых команд может быть сформировано указанием перед ней одного или нескольких присваиваний переменным. Так,

`TERM=d460 <команда>`

и

`(export TERM; TERM=d460; <команда>)`

эквивалентны. Переменные, участвующие в таких присваиваниях, назовем ключевыми параметрами.

Если установлен флаг `-k` (см. `set`), то все ключевые параметры помещаются в окружение команды, даже если они записаны после команды.

### **Зарезервированные слова**

Следующие слова являются зарезервированными:

```
if   then  else  elif  fi
case in    esac { }
for  while until do  done
```

### **Условные выражения**

Ветвление вычислительного процесса осуществляется с помощью оператора `if`:

```
if список_команд1
then список_команд2
[else список_команд3]
fi
```

*Список\_команд* - это одна или несколько команд (для задания пустого списка используется двоеточие - `:'`). *Список\_команд1* передает оператору `if` код возврата последней команды из списка. Если он равен 0, то выполняются команды из *списка\_команд2*, таким образом нулевой код воз-

врата эквивалентен значению «истина». В противном случае выполняются команды из *списка\_команд3*, если он существует.

Проверка условия может осуществляться с помощью команды **test**. Аргументами этой команды могут быть имена файлов, числовые и нечисловые строки. Она используется в следующих режимах:

- проверка файлов: **test** -ключ имя\_файла  
 Ключи:    -r     файл существует и доступен для чтения;  
           -w     файл существует и доступен для записи;  
           -x     файл существует и доступен для исполнения;  
           -f     файл существует и имеет тип '-', т.е. это обычный файл;  
               -s     файл существует, имеет тип '-' и не пуст;  
               -d     файл существует и имеет тип 'd', т.е. это каталог.
- сравнение чисел: **test** число1 -ключ число2  
 Ключи:    -eq    равно;  
           -ne    не равно;  
           -lt    меньше;  
           -le    меньше или равно;  
           -gt    больше  
           -ge    больше или равно.
- сравнение строк:  
**test [-n] строка**               строка непуста;  
**test -z строка**               строка пуста;  
**test строка1 = строка2** строки равны;  
**test строка1 != строка2**       строки не равны.

Пример 1

```
if test -w $2 -a -r $1
then cat $1 >> $2
else echo "cannot append"
fi
```

Пример 2

```
echo "Vvedite a"
read a
echo "Vvedite b"
read b
if test $a -lt $b
then
echo "Hello"
else
echo "..."
```



fi

### Построение циклов

В языке shell есть три типа циклов: *while*, *until* и *for*.

- цикл **while**:

```
while список_команд1{;/перевод строки}
do список_команд2{;/перевод строки}
done
```

В условии учитывается код возврата последней выполненной команды из *списка\_команд1*, при этом 0 интерпретируется как «истина».

- цикл **until**:

```
until список_команд1{;/перевод строки}
do список_команд2{;/перевод строки}
done
```

Проверка условия выполняется перед выполнением цикла. Учитывается код возврата последней выполненной команды из *списка\_команд1*, при этом цикл выполняется до тех пор, пока код возврата не примет значение «истина».

- цикл **for**:

```
for переменная [in список_значений]{;/перевод строки}
do список_команд{;/перевод строки}
done
```

Переменной присваивается значение очередного слова из *списка\_значений* и для этого значения выполняется *список\_команд*. Количество итераций равно количеству цепочек символов в *списке\_значений*, разделенных пробелами. Если слово *in* и *список\_значений* опущены как необязательные, то переменной поочередно присваиваются значения параметров, переданных при запуске данной программы. В качестве списка можно использовать шаблоны имен файлов, тогда интерпретатор превращает этот шаблон в требуемый синтаксисом список имен файлов, удовлетворяющий шаблону.

*Пример.* Печать списка имен текстовых файлов из текущего каталога.

```
list=`ls *.txt`
for val in $list
do
    echo «$val»
```

done

## **Vi. Текстовый редактор**

Запуск редактора: vi [+n] имя\_файла

+ вывести на экран конец файла;

n вывести на экран текст файла, начиная со строки n.

Текстовый полноэкранный редактор vi работает в двух основных режимах: в режиме “ввод текста” и в режиме “команда”.

Режим “ввод текста”. В этот режим редактор переводится с помощью клавиш <a> и <i>:

<a> набор текста в текущую строку;

< i > вставка текста в текущую строку перед курсором;

<ESC> выход из режима “ввод текста” в режим “команда”.

Режим “команда”. Это - основной режим редактирования текста:

<x> уничтожение текущего символа, выделенного курсором;

<r> замена текущего символа на символ, набранный вслед за

командой <r>;

<s> замена одного или нескольких символов текстом.

Например: 2sTEXT - замена двух текущих символов на слово TEXT;

<o> вставить пустую строку после текущей;

[n]<dw> уничтожить текущее слово или n слов;

[n]<dd> уничтожить текущую строку или n строк.

Выход из редактора

<ESC>:wq! Выход с сохранением;

<ESC>:q! Выход без сохранения.

### **Задания на лабораторную работу**

1. Подсчитать количество строк, содержащих заданное слово в заданном файле. Если файл имеет тип, отличный от ".txt", подсчет не производить, а просто вывести сообщение об этом.
2. В заданном файле определить повторяющиеся строки, вывести их номера.
3. Определить, кто из пользователей с первыми буквами имени "st" вошел в систему раньше всех. Послать ему сообщение, тело которого состоит из 2-х строк всех заданных файлов.
4. В заданном каталоге определить, какие имена файлов являются жесткими ссылками на один и тот же файл.
5. В заданном каталоге найти пустые файлы. Создать в своем домашнем каталоге одноименные файлы и занести в них содержимое файла-параметра.
6. Вывести имена всех файлов и подкаталогов в заданном каталоге, принадлежащих пользователю с заданным именем и общее число всех остальных файлов и каталогов.
7. Вывести списки всех файлов в заданном каталоге, которые были созданы в один день.
8. Вводятся целочисленные значения двух переменных. Вводится диапазон данных. Пока значения переменных находятся в указанном диапазоне, их значения инкрементируются.

## Литература

1. Дансмур М., Дейвис Г. Операционная система UNIX и программирование на языке Си: Пер. с англ.- М.: "Радио и связь", 1989.-192 с.
2. Забродин Л.Д. UNIX. Введение в командный интерфейс. -М.: "ДИАЛОГ-МИФИ", 1994.-144 с.
3. Керниган Б.В., Пайк Р. UNIX - универсальная среда программирования: Пер. с англ.- М.: Финансы и статистика, 1992.-304 с.
4. Робачевский А.М. Операционная система UNIX. - СПб.: BHV - Санкт-Петербург, 1997. - 528 с.
5. Т. Чан Системное программирование на C++ для UNIX. /Пер. с англ. -К.: Издательская группа BHV, 1997. - 592 с.

## Приложение А

### Команды Shell

#### Информационные команды

info имя...

man [раздел] имя...

#### Ввод и редактирование текстов

Дублирование стандартного вывода

tee [опции] [файл]...

#### Интерактивный текстовый редактор

ed [опции] [файл]...

#### Потоковый текстовый редактор

sed [опции] [файл]...

#### Полноэкранный текстовый редактор

vi [опции] [файл]...

#### Вывод текстов

Конкатенация файлов

cat [файл...]

#### Позкранный просмотр текста

more [-cdf] [-n] [+/-шаблон] [+n] [файл ...]

аргументы:

- n     печать в n строк
- +n     печать с n-й строки
- c     вывод с предварительной очисткой экрана

#### Печать файлов

pr [аргумент]... [файл]...

аргументы:

- n     печать в n колонок
- +n     печать с n-й страницы
- T     подавление печати заголовка и т.п.
- ln     длина страницы

#### Просмотр файлов на экране

pg [аргумент]... [файл]...

**Команды файловой системы**

Смена группы файла

cd каталог

Смена группы файла

chgrp группа файл

**Установка кода защиты файла**

chmod код\_защиты файл

код\_защиты:

0400 чтение для владельца

0200 запись для владельца

0100 выполнение для владельца

0700 чтение, запись, выполнение для группы

0007 чтение, запись, выполнение для прочих

**Смена владельца файла**

chown имя файл

**Копирование файла**

cp файл\_1 файл\_2

cp файл... каталог

**Тип файла**

file имя...

**Поиск файлов**

find список\_поиска выражение

аргументы:

-name имя файла

mtime n файлы изменены в течение n последних дней

-print печать найденных имен файлов

**Создание ссылки**

ln [-s] файл ссылка -s создание символической ссылки

**Содержимое каталогов**

ls [-опции...] имя...

опции:

l полная информация

t сортировка по времени

a вывод всех имен (. и ..)

**d** информация о каталогах

### **Создание каталогов**

**mkdir** имя...

### **Переименование/перенос файлов**

**mv** файл... целевой\_файл

### **Текущий каталог**

**pwd**

### **Удаление файлов**

**rm** [опции] файл...

опции:

- b** безусловное удаление
- r** удаление всех файлов и подкаталогов
- i** интерактивное удаление

### **Удаление каталогов**

**rmdir** каталог...

### **Обработка [текстовых] файлов**

Сопоставление с шаблонами и преобразование текста

**awk** [-Fсимвол] [[-f] программа] ['скрипт awk'] [файл ...]

- F** задание символа-разделителя
- f** чтение сценария awk из файла

### **Сравнение двух файлов**

**cmp** [-l][-s] файл\_1 файл\_2 **-l** печать полной таблицы различий

- s** установка кода возврата

### **Построчное сравнение файлов**

**comm** [-[123]] файл\_1 файл\_2 **123** номера печатаемых колонок

### **Удаление секции из каждой строки файла**

- cut** [cfd] [файл...] **-f** - задание списка полей
- d** - задание разделителя

**Различия между двумя файлами**

diff [-efbh] файл\_1 файл\_2 -e генерация команд редакто-  
ра

- b игнорировать лишние пробелы
- f краткий список различий

**Различия между тремя файлами**

diff3 [-ex3] файл\_1 файл\_2 файл\_3 -e генерация команд ре-  
дактора

- x3 различия только для файл\_3

**Поиск строк по шаблону**

grep [опции] выражение... [файл]  
опции:

- v печать строк без шаблона
- n печать строк с номерами
- y сопоставление строчных и прописных букв

**Вывод первых строк файла**

head [-cn] [файл...]  
-n число строк  
-c число байт

**Соединение файлов**

join [ajt] файл1 файл2  
-a печатать непарные строки  
-i игнорировать регистр  
-j задание номера поля  
-t задание разделителя

**Сцепление строк файлов**

paste [-ds] [файл...]  
-d замена символа табуляции  
-s сцепление последовательных строк

**Сортировка**

sort [-опции] [+pos] [-pos2]... [-o имя] [имя]...  
опции:

- m слияние
- n арифметическая сортировка
- o имя выходного файла
- u игнорировать одинаковые строки



**Разбиение файла**

split [-n] [файл [имя]] -n число строк в выходных файлах

**Вывод последних строк файла**

tail [+] [-][число][f] [файл...] f отслеживание приращения файла

**Вывод одинаковых строк файла**

uniq [-опции] [файл\_1 [файл\_2]]

опции:

u вывод неповторяющихся строк  
d вывод повторяющихся строк

**Подсчет числа слов**

wc [-lwc] [файл...]

l число строк

w число слов

c число символов

**Почта, процессы, время, etc.****Календарь**

cal [-mju] [[месяц] год] -m Monday - 1-й день недели

-j отображение порядкового номера дня

-y календарь на текущий год

**Печать и установка времени**

date [yymmddhhmm[.ss]]

**Вывод аргументов**

echo [-n] [аргумент] -n отмена перевода строки

**Чтение окружения**

env

**Добавление переменных в окружение**

export [переменная[=значение]]...

**Принудительное прекращение процесса**

kill ID\_процесса...

**Почта**

mail [имя...]

mail [опции...]

**Запрет выдачи сообщений на терминал**

mesg [опции...]

опции:

n     запретить сообщения  
 y     разрешить сообщения

**Понижение приоритета команды**

nice [-число] [команда [аргументы]]

**Состояние процессов**

ps [опции...] [имя]

опции:

a     все терминальные процессы  
 e     все процессы  
 l     полная информация

**Протоколирование сеанса**

script [-a] файл

**Вызов интерпретатора shell**

sh файл

**Информация о работающих пользователях**

who [файл] [aml]

**Передача сообщения другому пользователю**

write адресат

**Команды интерпретатора shell****Выделение локального имени**

basename имя\_файла

**Вычисление выражения**

expr выражение

**Ввод**

read

**Сдвиг списка параметров**

shift

**Проверка условия**

test выражение

Министерство образования и науки Российской Федерации

Федеральное агентство по образованию  
Государственное образовательное учреждение  
высшего профессионального образования  
«Комсомольский-на-Амуре Государственный технический универси-  
тет»  
Кафедра «Электропривод и автоматизация промышленных устано-  
вок»

### **Файловая система ОС UNIX**

Методические указания к лабораторной работе по курсу  
«СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ» для студентов направ-  
ления 220200.62 «Управление в технических системах для всех форм обу-  
чения

Комсомольск-на-Амуре 2011

УДК 004.4:004.31-181.4;004.318-181.4:004.4

Файловая система ОС UNIX: методические указания к лабораторной работе по курсам: «Системное программное обеспечение» /сост.: Черный С.П., Петренко Е.Д., Мешков А.С., – Комсомольск-на-Амуре : ГОУВПО «КнАГТУ», 2011. – 16 с.

Методические указания посвящены изучению принципов организации файловой системы UNIX и приобретению навыков написания программ работы с файлами.

Предназначены для студентов специальности 220201 «Управление и информатика в технических системах для всех форм обучения.

Печатается по постановлению редакционно-издательского совета ГОУВПО «Комсомольский-на-Амуре государственный технический университет».

Согласованно с патентно-информационным отделом.

Рецензент Соловьев В.А.

### **Цель работы:**

Ознакомиться с файловой системой ОС UNIX, механизмами ее функционирования, основными элементами файловой системы: суперблок, описатели файлов, типы файлов, список свободных описателей файлов, список свободных блоков.

### **Теоретические сведения**

#### **Общие положения**

Файловая система - это ключевое звено, обеспечившее успешное применение UNIX. Основной особенностью файловой системы UNIX является то, что все, с чем работает ОС UNIX, она воспринимает в виде файла. Таким образом, файл является единой универсальной структурой данных, в рамках которой реализуется любая операция ввода-вывода.

Всякий файл ОС UNIX в соответствие с его типом может быть отнесен к одной из следующих четырех групп: обычные файлы, каталоги, специальные файлы, каналы.

**Обычный файл** представляет собой совокупность блоков диска, входящих в состав файловой системы ОС UNIX. В указанных блоках может быть произвольная информация.

**Каталоги** представляют собой файлы особого типа, отличающиеся от обычных прежде всего тем, что осуществить запись в них может только ядро ОС UNIX, в то время как доступ по чтению может получить любой пользовательский процесс, имеющий соответствующие полномочия. Каждый элемент каталога состоит из двух полей: поля имени файла и поля, содержащего указатель на описатель файла, где хранится вся информация о файле: дата создания, размер, код защиты, имя владельца и т.д. В любом каталоге содержится, по крайней мере, два элемента, содержащие в поле имени файла имена "." и "..". Элемент каталога, содержащий в поле имени файла контекст ".", в поле ссылки содержит ссылку на описатель файла, описывающий этот каталог. Элемент каталога, содержащий в поле имени файла контекст "..", в поле ссылки содержит ссылку на описатель файла, в котором хранится информация о родительском каталоге текущего каталога.

**Специальные файлы** - это некоторые файлы, каждому из которых ставится в соответствие свое внешнее устройство, поддерживаемое ОС UNIX и имеющее специальную структуру. Его нельзя использовать для хранения данных, как обычный файл или каталог. В то же время над специальным файлом можно производить те же операции, что и над обычным файлом: открывать, вводить и/или выводить информацию и т.д. Результат применения любой из этих операций зависит от того, какому конкретному

устройству соответствует обрабатываемый специальный файл, однако в любом случае будет осуществлена соответствующая операция ввода-вывода на внешнее устройство, которому соответствует выбранный специальный файл.

Четвертый вид файлов – каналы. **Канал** – это программное средство, связывающее процессы ОС UNIX буфером ввода/вывода

Файловая система ОС UNIX имеет иерархическую структуру, образующую дерево каталогов и файлов. Корневой каталог обозначается символом "/", путь по дереву каталогов состоит из имен каталогов, разделенных "/", например:

/usr/work/document

В каждый момент времени с любым пользователем связан текущий каталог, то есть местоположение пользователя в иерархической файловой системе.

### Системные функции ОС UNIX для работы с файловой системой

Возвращают дескриптор файла	<code>open, creat, dup, pipe, close</code>
Преобразуют имя в описание	<code>open, creat, chdir, chmod, stat, mkfifo, mound, mknod, link, unmount, unlink, chown</code>
Назначают inode	<code>creat, link, unlink, mknod</code>
Работают с атрибутами	<code>chown, chmod, stat</code>
Ввод/вывод из файла	<code>read, write, lseek</code>
Работают со структурой файловой системы	<code>mount, unmount</code>
Управляют деревьями	<code>chmod, chown</code>

### Управление файлами

ОС UNIX поддерживает операции ввода-вывода с помощью набора взаимосвязанных таблиц. Основной из них считается таблица описателей файлов (FDT). Она представляет собой хранящуюся в оперативной памяти ЭВМ структуру данных, элементами которой являются копии описателей файлов, к которым была осуществлена попытка доступа. Каждому элементу таблицы описателей файлов обязательно соответствует один или несколько элементов системной таблицы файлов (SFT). Элемент таблицы файлов содержит информацию о режиме открытия файла и положении указателя чтения-записи. Таким образом, каждый файл может быть одновременно открыт несколькими независимыми процессами, и при каждом

открытии файла количество элементов таблицы файлов будет увеличиваться на единицу.

Если процессы связаны родственными отношениями, то процесс-потомок, порожденный системным вызовом `fork`, унаследует все открытые файлы процесса-предка. Это осуществляется с помощью таблицы открытых файлов процесса, которая создается сразу после порождения процесса, содержит информацию только о файлах, открытых данным процессом и передается процессу-потомку в момент его порождения.

Для примера рассмотрим следующую последовательность системных вызовов:

```
fd1 = open("/etc/passwd", O_RDONLY);
```

```
fd2 = open("local", O_RDWR);
```

```
fd3 = open("/etc/passwd", O_WRONLY);
```

На Рис. 1 показана взаимосвязь между таблицей описателей файлов, таблицей файлов и таблицей открытых файлов процесса. Каждый вызов функции `open` возвращает процессу дескриптор файла, а соответствующая запись в таблице открытых файлов процесса указывает на уникальную запись в таблице файлов ядра, даже если один и тот же файл ("`/etc/passwd`") открывается дважды. Записи в таблице файлов для всех экземпляров одного и того же открытого файла указывают на одну запись в таблице описателей файлов, хранящихся в памяти. Процесс может обращаться к файлу "`/etc/passwd`" с чтением или записью, но только через дескрипторы файла, имеющие значения 3 и 5 (Рис.1). Ядро запоминает разрешение на чтение или запись в файл в строке таблицы файлов, выделенной во время выполнения функции `open`.

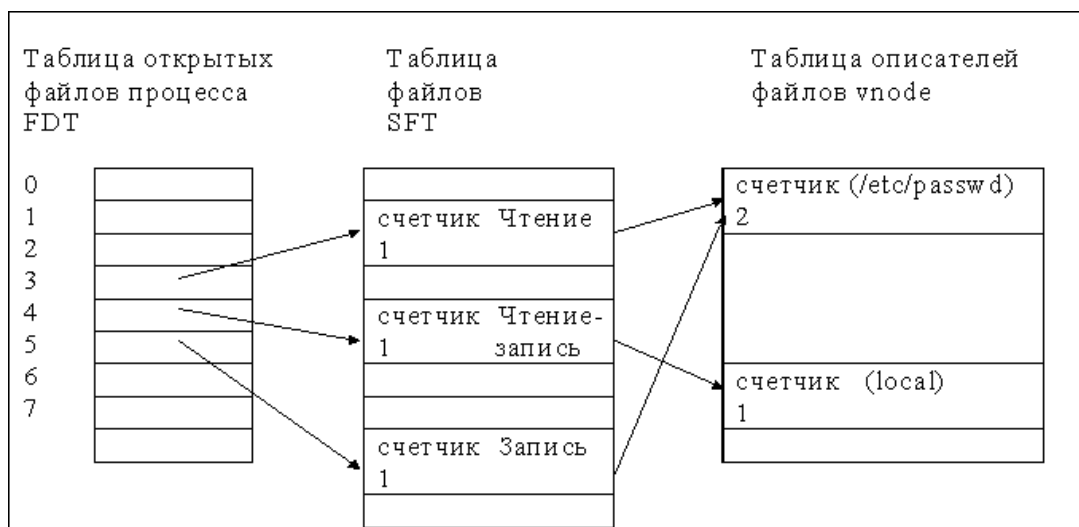


Рисунок - 1. Структуры данных после открытия файлов одним процессом



Предположим, что другой процесс выполняет следующий набор операторов:

```
fd1 = open("/etc/passwd",O_RDONLY);
```

```
fd2 = open("private",O_RDONLY);
```

На Рис. 2 показана взаимосвязь между соответствующими структурами данных, когда оба процесса (и больше никто) имеют открытые файлы. Снова результатом каждого вызова функции `open` является выделение уникальной точки входа в таблице открытых файлов процесса и в таблице файлов ядра, и ядро хранит не более одной записи на каждый файл в таблице описателей файлов, размещенных в памяти.

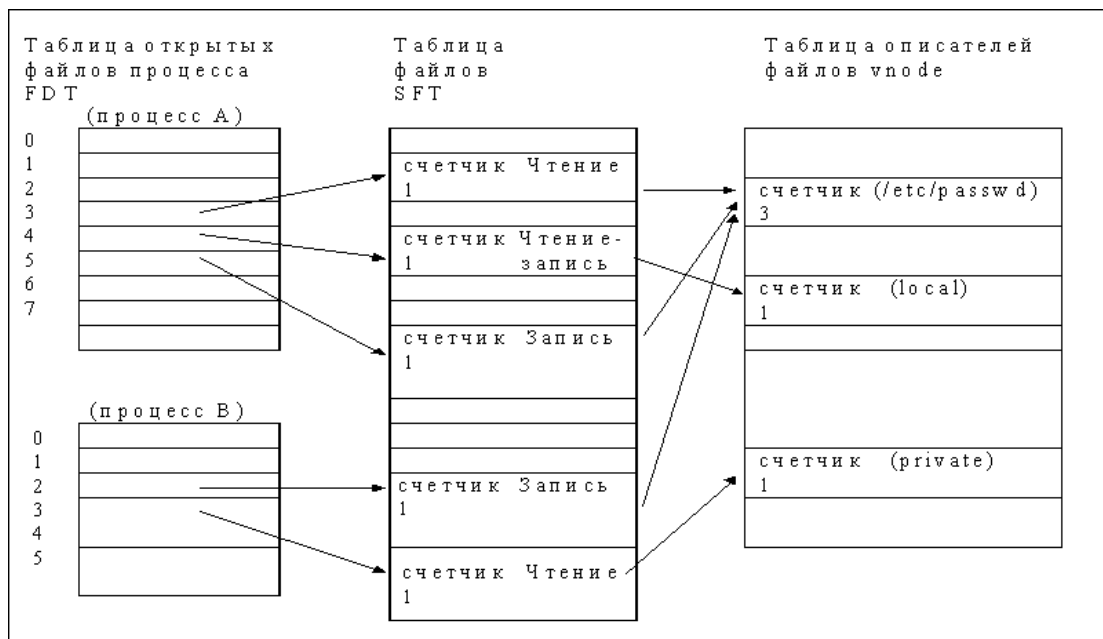


Рисунок - 2. Структуры данных после того, как два независимых процесса открыли файлы

Запись в таблице открытых файлов процесса по умолчанию хранит смещение в файле до адреса следующей операции ввода/вывода и указывает непосредственно на точку входа в таблице описателей для файла, устраняя необходимость в отдельной таблице файлов ядра.

Вышеприведенные примеры показывают взаимосвязь между записями таблицы открытых файлов процесса и записями в таблице файлов ядра типа “один к одному”. Однако, таблица файлов, реализованная как отдельная структура, позволяет совместно использовать один и тот же указатель смещения нескольким пользовательским дескрипторам файла. В системных вызовах `dup` (см. раздел “Программирование операций ввода-вывода”) и `fork` (лабораторная работа №3) при работе со структурами данных допускается такое совместное использование.

Первые три пользовательских дескриптора (0, 1 и 2) именуются дескрипторами файлов стандартного ввода, стандартного вывода и стандартного потока ошибок. Процессы в системе UNIX по договоренности используют дескриптор файла стандартного ввода при чтении вводимой информации, дескриптор файла стандартного вывода при записи выводимой информации и дескриптор стандартного файла ошибок для записи сообщений об ошибках. В операционной системе нет никакого указания на то, что эти дескрипторы файлов являются специальными. Группа пользователей может условиться о том, что файловые дескрипторы, имеющие значения 4, 6 и 11, являются специальными, но более естественно начинать отсчет с 0 (как в языке Си). Принятие соглашения сразу всеми пользовательскими программами облегчит связь между ними при использовании каналов.

Обычно операторский терминал служит и в качестве стандартного ввода, и в качестве стандартного вывода, и в качестве стандартного устройства вывода сообщений об ошибках.

Вызов `fork` порождает процесс, являющийся потомком по отношению к тому процессу, из которого осуществлен вызов. Процесс-потомок является точной копией процесса-предка за исключением номера самого процесса и значения, возвращаемого вызовом `fork`. При этом потомок получает к ранее открытым файлам доступ того же типа, что и предок (говорят, что процесс наследует открытые файлы). Родственные процессы общаются с общим файлом через один указатель чтения/записи, и если один из процессов прочитал или записал данные в файл, то значение указателя чтения/записи изменится для всех родственных процессов, имеющих доступ к этому файлу. Естественно, это не относится к файлам, которые были открыты родственными процессами после вызова `fork`: в этом случае каждый процесс обращается к файлу через собственный указатель.

Для примера рассмотрим следующую последовательность системных вызовов и состояние файловых таблиц после их выполнения (Рис.3):

```
fd1 = open("/etc/passwd", O_RDONLY);
pid = fork();
fd2 = open("private", O_RDWR);
```

Первый вызов `open` выполняется до вызова `fork`, он создает записи, относящиеся к файлу `"/etc/passwd"`, во всех файловых таблицах. При выполнении вызова `fork` процесс-потомок получает копию таблицы открытых файлов процесса, а в записях таблиц файлов и описателей файлов счетчики ссылок на файл `"/etc/passwd"` увеличиваются на единицу и становятся равным 2.

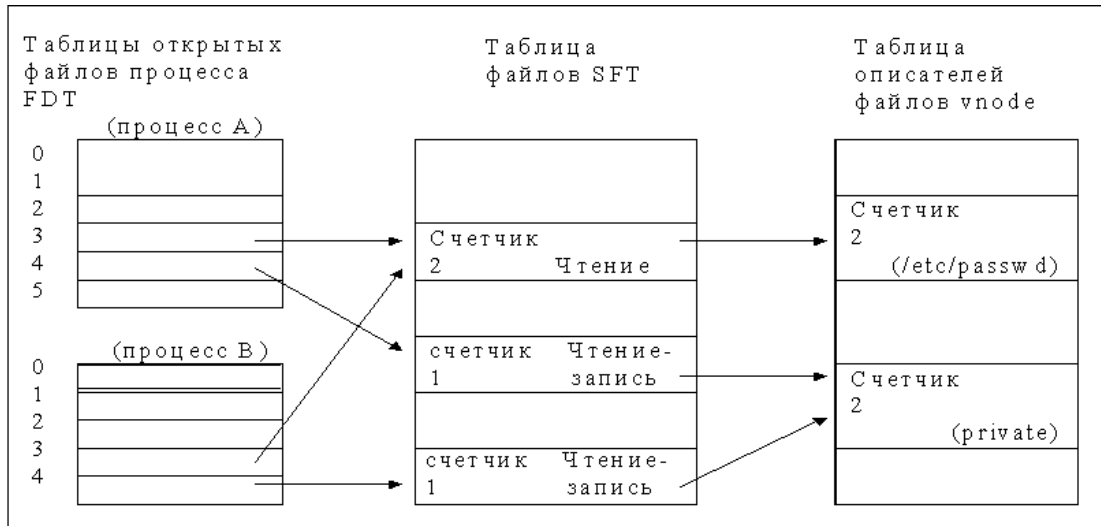


Рисунок - 3. Структуры данных после того, как два родственного процесса открыли файлы

Дополнительная запись в таблицу файлов не добавляется, оба процесса имеют доступ к файлу через один указатель чтения/записи. Второй вызов `open` выполняется после вызова `fork`, то есть тогда, когда существуют уже два процесса – предок и потомок. Каждый из них выполняет открытие файла независимо от другого, поэтому новые записи добавляются во все таблицы (запись в таблице описателей одна на файл, но счетчик ссылок на файл равен числу открывших файл процессов).

Заккрытие файла уменьшает число ссылок на файл, и только когда оно становится равным 0, происходит удаление соответствующих записей из таблиц.

## Программирование операций Ввода-Вывода

Системные вызовы представляют собой единственное средство, реализующее интерфейс между пользовательскими программами и ядром ОС UNIX. Всякая операция ввода/вывода для пользователя - это операция ввода/вывода в файл. Рассмотрим наиболее часто используемые из системных вызовов.

### OPEN

Открывает файл для получения доступа к нему:

```
int open(char *pathname, int flags, mode_t mode);
```

Возвращает положительное целое число, так называемый пользовательский дескриптор файла `fd`, который в дальнейшем используется для обращения к этому файлу. `pathname` - указатель на строку символов, содержащую полное имя файла. `mode` - режим открытия файла (по чтению,

записи и др.) Если нет возможности открыть файл, `open` возвращает -1. `flags` определяет режим открытия файла (`O_CREAT`, `O_TRUNC`, `O_RDONLY`, `O_WRONLY` и т.д.), `mode` задает права доступа к создаваемому файлу.

## CLOSE

Закрывает файл, уничтожает связь между пользовательским дескриптором файла и самим файлом:

```
void close(int fd);
```

Параметр `fd` - дескриптор файла, возвращенный вызовом `open`.

## STAT и FSTAT

Эти системные вызовы позволяют получить информацию о файле, не осуществляя явного доступа к нему:

```
int stat(char *path, struct stat *statbuf);
```

```
int fstat(int fd, struct stat *statbuf);
```

Оба системных вызова помещают информацию о файле (в первом случае специфицированным именем `path`, а во втором - дескриптором файла `fd`) в структурную переменную, на которую указывает `statbuf`. Вызывающая функция должна позаботиться о резервировании места для возвращаемой информации; в случае успеха возвращается 0, в противном случае -1 и код ошибки в `errno`. Описание структуры `stat` содержится в файле `<sys/stat.h>`.

```
struct stat
{
    dev_t st_dev;      /* device file */
    ino_t st_ino;      /* file serial inode */
    ushort st_mode;    /* режим доступа и тип файла */
    short st_nlink;    /* счетчик числа ссылок на файл */
    ushort st_uid;     /* идентификатор его владельца */
    ushort st_gid;     /* идентификатор группы */
    dev_t st_rdev;     /* тип устройства */
    off_t st_size;     /* размер файла в байтах */
    time_t st_atime;   /* дата последнего доступа */
    time_t st_mtime;   /* дата последней модификации */
    time_t st_ctime;   /* дата создания */
}
```

Для детализации информации в поле `st_mode` используются следующие макросы:

```
#define S_IFMT 0170000 /* тип файла */
#define S_IFDIR 0040000 /* каталог */
#define S_IFCHR 0020000 /* байт-ориентированный спец.
файл */
#define _IFBLK 0060000 /* блок-ориентированный спец.
файл */
#define S_IFREG 0100000 /* обычный файл */
#define S_IFIFO 0010000 /* дисциплина FIFO */
#define S_ISUID 04000 /* идентификатор владельца */
#define S_ISGID 02000 /* идентификатор группы */
#define S_ISVTX 01000 /* сохранить свопируемый текст
*/
#define S_IREAD 00400 /* владельцу разрешено чтение */
#define S_WRITE 00200 /* владельцу разрешена запись */
#define S_IEXEC 00100 /* владельцу разрешено исполне-
ние */
```

Пример использования вызова `stat`:

```
struct stat stbuf;
char *filename = "myfile";
.....
stat(filename, &stbuf);
if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
printf("%s является каталогом", filename);
```

## READ

Осуществляет чтение из открытого файла указанного количества символов в буфер:

```
int read(int fd, void *buffer, unsigned count);
```

Возвращает количество реально прочитанных байт `n` или отрицательный код ошибки.

## WRITE

Осуществляет запись в открытый файл указанного количества символов из буфера:

```
int write(int fd, void *buffer, unsigned count);
```

Возвращает количество реально записанных байт `n` или отрицательный код ошибки.

**LSEEK**

Перемещает указатель файла с пользовательским дескриптором fd на offset байт:

```
long lseek(int fd, long offset, int fromwhere);
```

Параметр fromwhere определяет положение указателя файла перед началом перемещения: SEEK\_SET - от начала файла;

SEEK\_CUR - от текущей позиции указателя;

SEEK\_END - от конца файла.

**DUP и DUP2**

Эти системные вызовы дублируют пользовательский дескриптор файла:

```
int dup(int handle);
```

```
int dup2(int oldhandle, int newhandle);
```

```
fd1 = dup(handle);
```

```
fd2 = dup2(oldhandle, newhandle);
```

Копия пользовательского дескриптора позволяет осуществлять к файлу доступ того же типа и с использованием того же указателя чтения/записи, что и с помощью оригинального дескриптора.

Вызов dup возвращает первый свободный номер дескриптора fd1 или -1, если указанный дескриптор handle не соответствует открытому файлу или нет свободных номеров.

Вызов dup2 возвращает дескриптор newhandle как копию дескриптора oldhandle или -1, если указанный дескриптор oldhandle не соответствует открытому файлу. Если newhandle до этого указывал на открытый файл, этот файл в результате вызова dup2 будет закрыт.

**Примеры программ работы с файлами**Пример 1. Запись в файл / чтение из файла.

Обратите внимание на обработку параметров командной строки.

```
/*
```

Фрагмент программы RW.C записи в файл / чтения из файла.

Программа воспринимает в качестве параметра командной строки имя рабочего файла. Если файл не существует, он будет создан, если существует, его содержимое будет изменено

```
*/
```

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

```
#include <sys/stat.h>
```

```
#include <sys/types.h>
```

```
#include <fcntl.h>
```

```

int fd;
int f1()
{
    static int j = 1;
    if (j > 10) return 0;
    write(fd, &j, sizeof(int));
    printf("write %d -- %d\n", fd, j++);
    return 1;
}
void f2()
{
    int i;
    lseek(fd, -sizeof(int), 1);
    read(fd, &i, sizeof(int));
    printf("read %d -- %d\n", fd, i);
}

void main(int argc, char *argv[])
{
    if (argc < 2) puts("Format: rw filename");
    else
    {
        fd = open(argv[1], O_CREAT | O_RDWR);
        while (f1()) f2();
        close(fd);
    }
    exit(0);
}

```

Пример 2. Дублирование дескриптора файла.

/\*

Фрагмент программы DUP.C - перенаправление стандартного вывода в файл.

\*/

```

#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>

```

```

void main(void)
{
    int outf, std_out;

```

```

char *str1 = "Вывод строки в файл ",
*str2 = "Вывод строки на экран";

std_out = dup(1);
/* закрытие стандартного вывода */
close(1);
outf = open("1.dat", O_WRONLY);
puts(str1);
write(std_out, str2, strlen(str2));

/* восстановление предыдущих значений */
close(outf);
outf = open("dev/tty", O_WRONLY);
close(std_out);
exit(0);
}

```

### **Выполнение лабораторной работы**

Выполнение работы заключается в написании и отладке программы по одному из вариантов задания. Ввод текста программы и его редактирование производится с помощью любого редактора UNIX (vi, ed и др.). Компиляция программы осуществляется с помощью следующего вызова:

```
$ cc имя_файла.c
```

На выходе получается исполняемый файл "a.out" или список сообщений об ошибках. Расширение указывать обязательно. Если запустить компилятор с опцией -o, можно указать произвольное имя исполняемого файла:

```
$ cc -o имя_исполняемого_файла имя_файла.c
```

Запуск исполняемого файла a.out

```
$ ./a.out
```



### Задания на лабораторную работу

1. Написать программу, меняющую в файле местами группы байт с 21-го по 28-й и с 33-го по 40-й. Имя файла вводится в командной строке.
2. Написать программу, переписывающую из входного файла каждый  $n$ -й байт в выходной файл. Имена входного и выходного файлов вводятся в командной строке.
3. Написать программу, переписывающую все байты входного файла в выходной файл в обратном порядке. Имена входного и выходного файлов вводятся в командной строке.
4. Написать программу, осуществляющую поиск заданного шаблона (последовательности символов) в файле. При обнаружении шаблона заменить его на последовательность символов с кодом 0 такой же длины, что и длина шаблона. Имя файла и шаблон вводятся в командной строке.
5. Написать программу, осуществляющую поиск в файле последовательностей, состоящих из двух и более пробелов, и удаление всех из них, кроме первого. Имя файла вводится в командной строке.
6. Написать программу, осуществляющую сравнение двух входных файлов. Результат работы программы выводится в выходной файл и состоит либо из сообщения о том, что расхождений в файлах нет, либо следующую диагностику:  
N  
  
имя\_вх\_файла\_1 фрагмент\_вх\_файла\_1  
имя\_вх\_файла\_2 фрагмент\_вх\_файла\_2 где  
N - номер байта, с которого начинается расхождение; фрагмент\_вх\_файла - 10 байт до первого расхождения и 10 байт после. Имена файлов вводятся в командной строке.
7. Написать программу кодировки входного файла на основании заданного кодового слова с возможностью декодирования (алгоритм сложения по модулю два). Имя входного файла и кодовое слово вводятся в командной строке.
8. Написать программу, осуществляющую подсчет количества строк в текстовом файле и запись полученного числа в начало этого файла первой строкой. Имя файла вводится в командной строке.
9. Написать программу, осуществляющую подсчет количества слов в текстовом файле и запись полученного числа в начало этого файла первой строкой. Имя файла вводится в командной строке.
10. Написать программу, осуществляющую замену в файле всех символов с кодами 0-31 на пробелы. Имя файла вводится в командной строке.
11. Написать программу, разбивающую текстовый файл на страницы по N строк, то есть добавляющую в файл после каждых N строк символ

перевода страницы (код 12). Имя файла и число N вводятся в командной строке.

12. Написать программу, переводящую текстовый файл из формата UNIX в формат DOS, то есть добавляющую после каждого символа перевода строки (код 10) символ возврата каретки (код 13). Имя файла вводится в командной строке.

13. Написать программу, выводящую в файл протокола список файлов указанной директории. Директория и имя файла протокола вводятся в командной строке; если имя файла не указано, список выводится на экран.

14. Написать программу, определяющую количество файлов в поддере каталогов, начиная с указанной директории. Имя директории вводится в командной строке.

15. Написать программу, устанавливающую биты разрешения доступа по исполнению каждому файлу в указанной директории, если для этого файла разрешено исполнение хотя бы для одной группы пользователей. Имя директории вводится в командной строке.

16. Написать программу, выводящую в файл протокола список файлов указанного каталога, созданных или модифицированных в текущий день. Имена протокола и каталога вводятся в командной строке.

17. Написать программу, выводящую в файл протокола список имен владельцев файлов в указанном каталоге. Имена файла протокола и каталога вводятся в командной строке.

18. Написать программу, выводящую в файл протокола список файлов из указанного каталога, имеющих n и более ссылок. Количество ссылок n, а также имена файла протокола и каталога вводятся в командной строке.

19. Написать программу, выводящую содержимое входного файла на экран или в выходной файл (если указано его имя), а сообщения об ошибках - в любом случае на экран, используя дублирование потоков (dup). Имена входного и выходного файла вводятся в командной строке.

20. Написать программу, вводящую N байт из стандартного входного потока или из входного файла (если указано его имя) и запрашивающую количество байт N с клавиатуры (с использованием дублирования потоков (dup)). Имя входного файла вводится в командной строке.

21. Написать программу, выводящую содержимое входного файла на экран и дублирующую протокол (stderr) на экран с использованием дублирования потоков (dup). Имя входного файла вводится в командной строке.

## Литература

1. Дансмур М., Дейвис Г. Операционная система UNIX и программирование на языке Си: Пер. с англ.- М.: "Радио и связь", 1989.-192 с.
2. Забродин Л.Д. UNIX. Введение в командный интерфейс. -М.: "ДИАЛОГ-МИФИ", 1994.-144 с.
3. Керниган Б.В., Пайк Р. UNIX - универсальная среда программирования: Пер. с англ.- М.: Финансы и статистика, 1992.-304 с.
4. Робачевский А.М. Операционная система UNIX. - СПб.: BHV - Санкт-Петербург, 1997. - 528 с.
5. Т. Чан Системное программирование на C++ для UNIX. /Пер. с англ. -К.: Издательская группа BHV, 1997. - 592 с.

Министерство образования и науки Российской Федерации

Федеральное агентство по образованию  
Государственное образовательное учреждение  
высшего профессионального образования  
«Комсомольский-на-Амуре Государственный технический универси-  
тет»  
Кафедра «Электропривод и автоматизация промышленных устано-  
вок»

### **Процессы и сигналы ОС UNIX**

Методические указания к лабораторной работе по курсу  
«СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ» для студентов направ-  
ления 220200.62 «Управление в технических системах для всех форм обу-  
чения

Комсомольск-на-Амуре 2011

УДК 004.4:004.31-181.4;004.318-181.4:004.4

Процессы и сигналы ОС UNIX: методические указания к лабораторной работе по курсам: «Системное программное обеспечение» /сост.: Черный С.П., Петренко Е.Д., Гудим А.С., Мешков А.С. – Комсомольск-на-Амуре : ГОУВПО «КНАГТУ», 2011. – 30 с.

Методические указания посвящены изучению создания процессов, их управлению и синхронизации, а также способам обмена данными и сигналами между процессорами в ОС UNIX.

Предназначены для студентов специальности 220201 «Управление и информатика в технических системах для всех форм обучения.

Печатается по постановлению редакционно-издательского совета ГОУВПО «Комсомольский-на-Амуре государственный технический университет».

Согласованно с патентно-информационным отделом.

Рецензент Соловьев В.А.

### **Цель работы:**

Изучить программные средства создания процессов, получить навыки управления и синхронизации процессов, а также простейшие способы обмена данными между процессами. Ознакомиться со средствами динамического запуска программ в рамках порожденного процесса, изучить механизм сигналов ОС UNIX, позволяющий процессам реагировать на различные события, и каналы, как одно из средств обмена информацией между процессами.

### **Теоретические сведения**

#### **Общие положения**

#### **Процессы ОС UNIX**

Работу ОС UNIX можно представить в виде функционирования множества взаимосвязанных процессов.

Процесс - это задание в ходе его выполнения. Он представляет собой исполняемый образ программы, включающий отображение в памяти исполняемого файла, полученного в ходе компиляции, стек, код и данные библиотек, а также ряд структур данных ядра, необходимых для управления процессом. Выполнение процесса заключается в точном следовании набору инструкций, который является замкнутым и не передает управление набору инструкций другого процесса. Он считывает и записывает информацию в раздел данных и в стек, но ему недоступны данные и стеки других процессов.

ОС UNIX является многозадачной системой, поэтому в ней параллельно выполняется множество процессов, их выполнение планируется ядром. Несколько процессов могут быть экземплярами одной программы. Процессы взаимодействуют с другими процессами и с вычислительными ресурсами только посредством обращений к операционной системе, которая эффективно распределяет системные ресурсы между активными процессами.

#### **Выполнение процесса**

Выполнение процесса осуществляется ядром. Подсистема управления процессами отвечает за синхронизацию процессов, взаимодействие процессов, распределение памяти и планирование выполнения процессов.

С практической точки зрения процесс в системе UNIX является объектом, создаваемым в результате выполнения системного вызова `fork`. Каждый процесс, за исключением нулевого, порождается в результате запуска другим процессом операции `fork`. Процесс, запустивший операцию `fork`, называется родительским, а вновь созданный процесс - порожденным.

Каждый процесс имеет одного родителя, но может породить много процессов. Ядро системы идентифицирует каждый процесс по его номеру, который называется идентификатором процесса (PID). Нулевой процесс является особым процессом, который создается «вручную» в результате загрузки системы. Процесс 1, известный под именем `init`, является предком любого другого процесса в системе и связан с каждым процессом.

Пользователь, транслируя исходный текст программы, создает исполняемый файл, который состоит из нескольких частей:

- набора «заголовков», описывающих атрибуты файла;
- текста программы;
- представления на машинном языке данных, имеющих начальные значения при запуске программы на выполнение, и указания на то, сколько пространства памяти ядро системы выделит под неинициализированные данные, так называемые `bss` («block started by symbol» - «блок, начинающийся с символа»);
- других секций, таких как информация символических таблиц.

Ядро загружает исполняемый файл в память при выполнении системной операции `exec`, при этом загруженный процесс состоит по меньшей мере из трех частей, так называемых областей: текста, данных и стека. Области текста и данных соответствуют секциям текста и `bss`-данных исполняемого файла, а область стека создается автоматически и ее размер динамически устанавливается ядром системы во время выполнения.

Процесс в системе UNIX может выполняться в двух режимах - режиме ядра или режиме задачи. В режиме задачи процесс выполняет инструкции прикладной программы, системные структуры данных ему недоступны.

Когда процесс выполняет специальную инструкцию (системный вызов), он переключается в режим ядра. Каждой системной операции соответствует точка входа в библиотеке системных операций; библиотека системных операций написана на языке ассемблера и включает специальные команды прерывания, которые, выполняясь, порождают «прерывание», вызывающее переключение аппаратуры в режим ядра. Процесс ищет в библиотеке точку входа, соответствующую отдельной системной операции, подобно тому, как он вызывает любую из функций.

Соответственно и образ процесса состоит из двух частей: данных режима ядра и режима задачи. В режиме ядра образ процесса включает сегменты кода, данных, библиотек и других структур, к которым он может получить непосредственный доступ. Образ процесса в режиме ядра состоит из структур данных, недоступных процессу в режиме задачи (например, состояния регистров, таблицы для отображения памяти и т.п.).

Каждому процессу соответствует точка входа (запись) в таблице процессов ядра, кроме того, каждому процессу выделяется часть оператив-

ной памяти, отведенной под задачи пользователей. Таблица процессов включает в себя указатели на промежуточную таблицу областей процессов, точки входа в которую служат в качестве указателей на собственно таблицу областей. Областью называется непрерывная зона адресного пространства, выделяемая процессу для размещения текста, данных и стека. Точки входа в таблицу областей описывают атрибуты области, как например, хранятся ли в области текст программы или данные, закрытая ли эта область или же совместно используемая, и где конкретно в памяти размещается содержимое области. Внешний уровень косвенной адресации (через промежуточную таблицу областей, используемых процессами, к собственно таблице областей) позволяет независимым процессам совместно использовать области.

Когда процесс запускает системную операцию `exec`, ядро системы выделяет области под ее текст, данные и стек, освобождая старые области, которые использовались процессом. Если процесс запускает операцию `fork`, ядро удваивает размер адресного пространства старого процесса, позволяя процессам совместно использовать области, когда это возможно, и, с другой стороны, производя физическое копирование. Если процесс запускает операцию `exit`, ядро освобождает области, которые использовались процессом. Таблица процессов ссылается на промежуточную таблицу областей, используемых процессом, в которой содержатся указатели на записи в собственно таблице областей, соответствующие областям для текста, данных и стека процесса.

UNIX является системой разделения времени, это означает, что каждому процессу вычислительные ресурсы выделяются на ограниченный промежуток времени, после чего они предоставляются другому процессу. Максимальный временной интервал, на который процесс может захватить процессор, называется временным квантом. Таким образом, создается иллюзия того, что процессы работают одновременно, хотя в действительности на однопроцессорной машине одновременно может выполняться только один процесс.

Процессы предъявляют различные требования к системе с точки зрения их планирования и общей производительности. Можно выделить три основных класса приложений:

- интерактивные приложения (командные интерпретаторы, редакторы и проч.). Большую часть времени они проводят в ожидании пользовательского ввода, но для них критично время отклика (реакции системы на ввод данных).
- фоновые приложения (не требующие вмешательства пользователя). Основной показатель эффективности для них - минимальное суммарное время выполнения в системе.



- приложения реального времени. Они обычно привязаны к таймеру и требуют гарантированного времени совершения той или иной операции и времени отклика.

В основе планирования выполнения процессов лежат два понятия: квант времени и приоритет. Каждый процесс имеет два атрибута приоритета: текущий (на основании которого осуществляется планирование) и относительный, который задается при порождении процесса и влияет на текущий. Номера приоритетов разбиваются на несколько групп: для процессов в режиме задачи, в режиме ядра, для процессов реального времени (группы указаны в соответствии с повышением приоритета).

Обработчик прерываний от таймера, в частности, проверяет истечение временного кванта для процессов и пересчитывает приоритеты процессов: чем дольше процесс занимает процессор, тем ниже (в пределах группы) становится его приоритет.

Выполнение процесса может быть прервано:

а) планировщиком процессов по истечении временного кванта или в том случае, если в очереди готовых к исполнению процессов есть процесс с более высоким приоритетом.

б) ядром системы, если процесс ожидает недоступного ресурса или окончания длительной операции ввода/вывода.

В режиме ядра приоритет процесса повышается для того, чтобы его выполнение не могло быть прервано, так как это может привести к нарушению целостности структур данных ядра. Таким образом, выполнение системных вызовов осуществляется в непрерывном режиме (за исключением некоторых аппаратных прерываний).

### **Контекст процесса**

Под контекстом процесса понимается вся информация, для описания процесса. Контекст процесса состоит из нескольких частей:

- адресное пространство процесса в режиме задачи (код, данные и стек процесса, а также разделяемая память и данные динамических библиотек);
- управляющая информация (структуры `proc` и `user` - запись таблицы процессов и дополнительная информация соответственно);
- окружение процесса (системные переменные, например, домашний каталог, имя пользователя и др.);
- аппаратный контекст (значения используемых машинных регистров).

Текст операций системы и ее глобальные информационные структуры совместно используются всеми процессами, но не являются составной частью контекста процесса. Принято говорить, что при запуске процесса система исполняется в контексте процесса. Когда ядро системы решает за-

пустить другой процесс, оно выполняет переключение контекста с тем, чтобы система исполнялась в контексте другого процесса. Ядро осуществляет переключение контекста только при определенных условиях. Выполняя переключение контекста, ядро сохраняет информацию, достаточную для того, чтобы позднее переключиться вновь на прерванный процесс и возобновить его выполнение.

Аналогичным образом, при переходе из режима задачи в режим ядра, ядро системы сохраняет информацию, достаточную для того, чтобы позднее вернуться в режим задачи и продолжить выполнение с прерванного места. Однако, переход из режима задачи в режим ядра является сменой режима, но не переключением контекста. Ядро меняет режим выполнения с режима задачи на режим ядра и наоборот, оставаясь в контексте одного процесса.

Ядро обрабатывает прерывания в контексте прерванного процесса, пусть даже оно и не вызывало никакого прерывания. Прерванный процесс мог при этом выполняться как в режиме задачи, так и в режиме ядра. Ядро сохраняет информацию, достаточную для того, чтобы можно было позже возобновить выполнение прерванного процесса, и обрабатывает прерывание в режиме ядра. Ядро не порождает и не планирует порождение какого-то особого процесса по обработке прерываний.

### **Состояния процесса**

Время жизни процесса можно разделить на несколько состояний, каждое из которых имеет определенные характеристики, описывающие процесс. Перечислим основные из состояний:

1. Процесс выполняется в режиме задачи.
2. Процесс выполняется в режиме ядра.
3. Процесс не выполняется, но готов к выполнению, находится в очереди готовых к исполнению процессов и ждет, когда планировщик выберет его. Естественно, в этом состоянии может находиться много процессов, и алгоритм планирования устанавливает, какой из процессов будет выполняться следующим.

4. Процесс приостановлен («спит»). Процесс «впадает в сон», когда он не может больше продолжать выполнение, например, когда ждет завершения ввода-вывода или освобождения какого-либо занятого ресурса.

Поскольку процессор в каждый момент времени выполняет только один процесс, в состояниях 1 и 2 может находиться самое большее один процесс.

Состояния процесса, перечисленные выше, дают статическое представление о процессе, однако процессы непрерывно переходят из состояния в состояние в соответствии с определенными правилами. Диаграмма переходов представляет собой ориентированный граф, вершины которого

представляют собой состояния, в которые может перейти процесс, а дуги - события, являющиеся причинами перехода процесса из одного состояния в другое. Переход между двумя состояниями разрешен, если существует дуга из первого состояния во второе. Несколько дуг может выходить из одного состояния, однако процесс переходит только по одной из них в зависимости от того, какое событие произошло в системе.

В режиме разделения времени может выполняться одновременно несколько процессов. Если им разрешить свободно выполняться в режиме ядра, то они могут испортить глобальные информационные структуры, принадлежащие ядру. Запрещая произвольное переключение контекстов и управляя возникновением событий, ядро защищает свою целостность. Ядро разрешает переключение контекста только тогда, когда процесс переходит из состояния «запуск в режиме ядра» в состояние «сна в памяти». Процессы, запущенные в режиме ядра, не могут быть выгружены другими процессами; поэтому иногда говорят, что ядро невыгружаемо, при этом процессы, находящиеся в режиме задачи, могут выгружаться системой. Ядро поддерживает целостность своих информационных структур, поскольку оно невыгружаемо, таким образом решая проблему «взаимного исключения» - обеспечения того, что критические секции программы выполняются в каждый момент времени в рамках самое большее одного процесса.

### **Сигналы как средство взаимодействия процессов**

**Сигнал** - это программное средство, с помощью которого может быть прервано функционирование процесса ОС UNIX. Сигналы сообщают процессам о возникновении асинхронных событий. Механизм сигналов позволяет процессам реагировать на различные события, которые могут происходить в ходе работы процесса внутри него самого или во внешней среде.

Сигналы описаны в файле <signal.h>, каждому из них ставится в соответствие мнемоническое обозначение. Количество и семантика сигналов зависят от версии ОС UNIX.

В версии System V сигналы имеют номера от 1 до 19:

```
#define NSIG          20
#define SIGHUP    1    /* разрыв связи */
#define SIGINT    2    /* прерывание */
#define SIGQUIT   3    /* аварийный выход */
#define SIGILL    4    /* неверная машинная инструкция */
#define SIGTRAP   5    /* прерывание-ловушка */
#define SIGIOT    6    /* прерывание ввода-вывода */
#define SIGEMT    7    /* программное прерывание EMT */
```

```

#define SIGFPE  8    /* авария при выполнении операции с */
/* плавающей точкой */
#define SIGKILL  9    /* уничтожение процесса */
#define SIGBUS   10   /* ошибка шины */
#define SIGSEGV  11   /* нарушение сегментации */
#define SIGSYS   12   /* ошибка выполнения системного вызова */
#define SIGPIPE  13   /* запись в канал есть, чтения нет */
#define SIGALRM   14   /* прерывание от таймера */
#define SIGTERM   15   /* программ. сигнал завершения от kill */
/*
#define SIGUSR1  16   /* определяется пользователем */
#define SIGUSR2  17   /* определяется пользователем */
#define SIGCLD   18   /* процесс-потомок завершился */
#define SIGPWR   19   /* авария питания */

#define SIG_DFL (int(*)())0 /* все установки «по умолчанию» */
#define SIG_IGN (int(*)())1 /* игнорировать этот сигнал */

```

В версии BSD UNIX сигналы описываются следующим образом:

```

#define SIGHUP   1    /* разрыв связи */
#define SIGINT   2    /* прерывание */
#define SIGQUIT  3    /* аварийный выход */
#define SIGILL   4    /* неверная машинная инструкция */
#define SIGTRAP  5    /* прерывание-ловушка */
#define SIGIOT   6    /* прерывание ввода-вывода */
#define SIGABRT  6    /* используется как ABORT */
#define SIGEMT   7    /* программное прерывание EMT */
#define SIGFPE   8    /* авария при выполнении операции
/* с плавающей точкой */
#define SIGKILL   9    /* уничтожение процесса (не может быть
/* перехвачен или проигнорирован */
#define SIGBUS   10   /* ошибка шины */
#define SIGSEGV  11   /* нарушение сегментации */
#define SIGSYS   12   /* неправильный аргумент системного
/* вызова */
#define SIGPIPE  13   /* запись в канал есть, чтения нет */
#define SIGALRM   14   /* прерывание от таймера */
#define SIGTERM   15   /* программ. сигнал завершения от kill */
/*
#define SIGUSR1  16   /* определяется пользователем */
#define SIGUSR2  17   /* определяется пользователем */

```

```

#define SIGCLD 18 /* изменение статуса потомка
(завершение процесса-потомка) */
#define SIGCHLD 18 /* альтернатива для SIGCLD (POSIX) */
#define SIGPWR 19 /* авария питания */
#define SIGWINCH 20 /* изменение размера окна */
#define SIGURG 21 /* urgent socket condition */
#define SIGPOLL 22 /* pollable event occurred */
#define SIGIO SIGPOLL /* socket I/O possible (SIGPOLL
alias) */
#define SIGSTOP 23 /* стоп (не может быть перехвачен или
проигнорирован) */
#define SIGTSTP 24 /* требование остановки от терминала */
#define SIGCONT 25 /* остановить процесс с возможностью
продолжения */
#define SIGTTIN 26 /* скрытая попытка чтения с терминала */
#define SIGTTOU 27 /* скрытая попытка записи на терминал */
#define SIGVTALRM 28 /* виртуальное время истекло */
#define SIGPROF 29 /* время конфигурирования истекло */
#define SIGXCPU 30 /* превышен лимит ЦП */
#define SIGXFSZ 31 /* превышен лимит размера файла */
#define SIGWAITING 32 /* process's lwps заблокирован */
#define SIGLWP 33 /* спецсигнал (used by thread library) */
#define SIGFREEZE 34 /* спецсигнал, используемый
процессором */
#define SIGTHAW 35 /* спецсигнал, используемый
процессором */
#define _SIGRTMIN 36 /* первый (с высшим приорите-
том)
сигнал реального времени */
#define _SIGRTMAX 43 /* последний (с низшим приори-
тетом)
сигнал реального времени */
#define SIG_DFL (void(*)())0 /* все установки «по умолчанию» */
#define SIG_IGN (void(*)())0 /* игнорировать этот сигнал */

```

Примечание: причины возникновения сигналов для различных версий могут отличаться; первоначально они были обусловлены архитектурными особенностями ЭВМ PDP-11.

### Причины возникновения сигналов

В версии System V UNIX возникновение сигналов можно классифицировать следующим образом:

- введение пользователем управляющего символа с терминала всем процессам, ассоциированным с данным терминалом (SIGINT, SIGQUIT, SIGHUP);
- возникновение аварийной ситуации при функционировании пользовательского процесса (SIGILL, SIGTRAP, SIGFPE, SIGBUS, SIGSEGV, SIGSYS, SIGPIPE);
- возникновение непредусмотренного или не поддающегося идентификации события (SIGTERM, SIGCLD, SIGPWR);
- возникновение некоторого заранее описанного события (SIGALRM).

Посылка сигналов производится процессами - друг другу, с помощью функции kill, - или ядром. Для каждого процесса определен бинарный вектор, длина которого равна количеству сигналов в системе. При получении процессом сигнала I соответствующий i-й разряд этого вектора становится равным 1. Каждому сигналу соответствует адрес функции, которая будет вызвана для обработки данного сигнала.

### **Обработка сигналов**

Ядро обрабатывает сигналы в контексте того процесса, который получает их, поэтому чтобы обработать сигналы, нужно запустить процесс.

Существует три способа обработки сигналов:

- реакция по умолчанию,
- игнорирование сигнала,
- выполнение особой (пользовательской) функции по его получении.

Реакцией по умолчанию со стороны процесса, исполняемого в режиме ядра, обычно является вызов функции exit(), т.е. завершение процесса. Но вместе с тем реакция процесса на принимаемый сигнал зависит от того, как сам процесс определил свое поведение в случае приема данного сигнала: процесс может проигнорировать сигнал, вызвать на выполнение другой процесс и т.д. При этом способ обработки сигналов одного типа не влияет на обработку сигналов других типов.

Обработывая сигнал, ядро определяет тип сигнала и очищает (гасит) разряд в записи таблицы процессов, соответствующий данному типу сигнала и установленный в момент получения сигнала процессом. Таким образом, когда процесс получает любой неигнорируемый им сигнал (за исключением SIGILL и SIGTRAP), ОС UNIX автоматически восстанавливает реакцию «по умолчанию» на всякое последующее получение этого сигнала.

Замечание 1: Если необходима многократная обработка одного и того же сигнала, процесс должен каждый раз осуществлять системный вызов `signal` для установления требуемой реакции на данный сигнал.

Замечание 2: Процесс не в состоянии узнать, сколько однотипных сигналов им было получено. В том случае, если процесс не успевает обработать все поступившие сигналы, происходит потеря информации.

Если функции обработки сигнала присвоено значение по умолчанию, ядро в отдельных случаях перед завершением процесса сбрасывает на внешний носитель (дампирует) образ процесса в памяти. Дампирование удобно для программистов тем, что позволяет установить причину завершения процесса и посредством этого вести отладку программ. Ядро дампирует состояние памяти при поступлении сигналов, которые сообщают о каких-нибудь ошибках в выполнении процессов, как например, попытка исполнения запрещенной команды или обращение к адресу, находящемуся за пределами виртуального адресного пространства процесса. Ядро не дампирует состояние памяти, если сигнал не связан с программной ошибкой, за исключением внешнего сигнала о выходе (`quit`), обычно вызываемого одновременным нажатием клавиш `Ctrl+|`.

Если процесс получает сигнал `SIGINT`, который было решено игнорировать (`signal(SIGINT, SIG_IGN)`), выполнение процесса продолжается так, словно сигнала и не было. Поскольку ядро не сбрасывает значение соответствующего поля, свидетельствующего о необходимости игнорирования сигнала данного типа, то когда сигнал поступит вновь, процесс опять не обратит на него внимание.

В том случае, если процесс получает сигнал, реагирование на который установлено системным вызовом `signal`, сразу по возвращении процесса в режим задачи выполняется заранее условленное действие, описанное в вызове `signal`. После выполнения функции обработки сигнала управление будет передано на то место в программе пользователя, где было произведено обращение к системной функции или произошло прерывание.

Если во время исполнения системной функции приходит сигнал, а процесс приостановлен с допускающим прерывания приоритетом, этот сигнал побуждает процесс выйти из приостанова, вернуться в режим задачи и вызвать функцию обработки сигнала. Когда функция обработки сигнала завершается, процесс выходит из системной функции с ошибкой, сообщающей о прерывании ее выполнения. После этого пользователь может запустить системную функцию повторно.

### **Группы процессов**

Несмотря на то, что в системе UNIX процессы идентифицируются уникальным кодом (`PID`), системе иногда приходится использовать для идентификации процессов номер «группы», в которую они входят. Напри-

мер, процессы, имеющие общего предка в лице регистрационного интерпретатора shell, взаимосвязаны, и поэтому когда пользователь нажимает клавиши «delete» или «break», или когда терминальная линия «зависает», все эти процессы получают соответствующие сигналы. Ядро использует код группы процессов для идентификации группы взаимосвязанных процессов, которые при наступлении определенных событий должны получать общий сигнал. Код группы запоминается в таблице процессов. При выполнении функции fork процесс-потомок наследует код группы своего родителя.

Для того, чтобы образовать новую группу процессов, следует воспользоваться системной функцией setpgrp:

```
grp = setpgrp();
```

где grp - новый код группы процессов, равный его коду идентификации процесса, осуществившего вызов setpgrp().

## СИСТЕМНЫЕ ВЫЗОВЫ

### Системные вызовы для работы с процессами

Рассмотрим системные вызовы, используемые при работе с процессами в ОС UNIX, описанные в библиотеке <fcntl.h>.

Системный вызов fork создает новый процесс, копируя вызывающий; вызов exit завершает выполнение процесса; wait дает возможность родительскому процессу синхронизировать свое продолжение с завершением порожденного процесса, а sleep приостанавливает на определенное время выполнение процесса. Системный вызов exec дает процессу возможность запускать на выполнение другую программу.

### FORK

Создание нового процесса:

```
int fork(void);
```

```
pid = fork();
```

Системный вызов **fork** служит для создания нового процесса. Новый процесс (процесс-потомок) является полной копией процесса-предка, за исключением того, что процесс-потомок имеет свой уникальный идентификатор процесса (PID). Поскольку новый процесс сам может выступать в качестве порождающего, его идентификатор процесса-предка (PPID), естественно, отличается от соответствующего идентификатора породившего его процесса.

Новый процесс имеет собственную копию таблицы дескрипторов открытых файлов, но эти дескрипторы ссылаются на те же самые файлы, что и дескрипторы процесса-предка, и имеют с ним общие указатели позиций чтения/записи. Так, при выполнении операции **lseek** в порожденном процессе, может измениться и позиция чтения/записи файла в процессе-



предке. Это свойство, в частности, используется командными языками SH CSH для переопределения файлов стандартного ввода/вывода и организации конвейеров команд.

В ходе выполнения функции ядро производит следующую последовательность действий:

1. Отводит место в таблице процессов под новый процесс.
2. Присваивает порождаемому процессу уникальный код идентификации.
3. Делает копию контекста родительского процесса. Поскольку те или иные составляющие процесса, такие как область команд, могут разделяться другими процессами, ядро может иногда вместо копирования области в новый физический участок памяти просто увеличить значение счетчика ссылок на область.
4. Увеличивает значения счетчика числа файлов, связанных с процессом, как в таблице файлов, так и в таблице индексов.
5. Возвращает родительскому процессу код идентификации порожденного процесса, а порожденному процессу - 0.

В результате выполнения функции `fork` пользовательский контекст и того, и другого процессов совпадает во всем, кроме возвращаемого значения переменной `pid`. Если процесс не может быть порожден, функция возвращает отрицательный код ошибки.

Процесс, вызывающий функцию `fork`, называется родительским (процесс-родитель или процесс-предок), вновь создаваемый процесс называется порожденным (процесс-потомок).

Процесс-потомок всегда имеет более высокий приоритет, чем процесс-предок, так как приоритет процесса является самым высоким в момент порождения и уменьшается по мере нахождения в состоянии выполнения.

Нулевой процесс, возникающий внутри ядра при загрузке системы, является единственным процессом, не создаваемым с помощью функции `fork`.

## EXIT

Завершение выполнения процесса:

```
void exit(int status);
```

где `status` - значение, возвращаемое функцией родительскому процессу. Процессы могут вызывать функцию `exit` как в явном, так и в неявном виде (по окончании выполнения программы функция `exit` вызывается автоматически с параметром 0). Также ядро может вызывать функцию `exit` по своей инициативе, если процесс не принял посланный ему сигнал. В этом случае значение параметра `status` равно номеру сигнала. Выполнение

вызова `exit` приводит к «прекращению существования» процесса, освобождению ресурсов и ликвидации контекста.

## **WAIT**

Ожидание завершения выполнения процесса-потомка:

```
int wait(int *stat);
```

```
pid = wait(stat_addr);
```

где `pid` - значение кода идентификации (PID) завершившегося потомка, `stat_addr` - адрес переменной целого типа, в которую будет помещено возвращаемое функцией `exit` значение.

С помощью этой функции процесс синхронизирует продолжение своего выполнения с моментом завершения потомка. Ядро ведет поиск потомков процесса, прекративших существование, и в случае их отсутствия возвращает ошибку.

Если потомок, прекративший существование, обнаружен, ядро передает его код идентификации и значение, возвращаемое через параметр функции `exit`, процессу, вызвавшему функцию `wait`. Таким образом, через параметр функции `exit` (`status`) завершающийся процесс может передавать различные значения, в закодированном виде содержащие информацию о причине завершения процесса, однако на практике этот параметр используется по назначению довольно редко.

Если процесс, выполняющий функцию `wait`, имеет потомков, продолжающих существование, он приостанавливается до получения ожидаемого сигнала. Ядро не возобновляет по своей инициативе процесс, приостановившийся с помощью функции `wait`: такой процесс может возобновиться только в случае получения сигнала о «гибели потомка».

## **SLEEP**

Приостанов работы процесса на определенное время:

```
void sleep(unsigned seconds);
```

где `seconds` - количество секунд, на которое требуется приостановить работу процесса.

Сначала ядро повышает приоритет работы процесса так, чтобы заблокировать все прерывания, которые могли бы (путем создания конкуренции) помешать работе с очередями приостановленных процессов, и запоминает старый приоритет, чтобы восстановить его, когда выполнение процесса будет возобновлено. Процесс получает пометку «приостановленного», адрес приостанова и приоритет запоминаются в таблице процессов, а процесс помещается в хеш-очередь приостановленных процессов. В простейшем случае (когда приостанов не допускает прерываний) процесс выполняет переключение контекста и благополучно «засыпает». Когда приостановленный процесс «пробуждается», ядро начинает планировать его

запуск: процесс возвращает сохраненный в алгоритме sleep контекст, восстанавливает старый приоритет работы процесса (который был у него до начала выполнения алгоритма) и возвращает управление ядру. Таким образом, нельзя гарантировать, что по истечении заданного времени приостановленный процесс сразу возобновит свою работу: он может быть выгружен на время приостанова и тогда требуется его подкачка в память; в это время на выполнении может находиться процесс с более высоким приоритетом или процесс, не допускающий прерываний (например, находящийся в критическом интервале) и т.д.

Параметр seconds устанавливает минимальный интервал, в течение которого процесс будет приостановлен, а реальное время приостанова в любом случае будет несколько больше, хотя бы за счет времени, необходимого для переключения процессов.

## **EXEC**

Запуск программы.

Системный вызов exec осуществляет несколько библиотечных функций - execl, execv, execl и др. Приведем формат одной из них:

```
int execv(char *path, char *argv[]);
res = execv(path, argv);
```

где path - имя исполняемого файла, argv - указатель на массив параметров, которые передаются вызываемой программе. Этот массив аналогичен параметру argv командной строки функции main. Список argv должен содержать минимум два параметра: первый - имя программы, подлежащей выполнению (отображается в argv[0] функции main новой программы), второй - NULL (завершающий список аргументов).

Системный вызов exec дает возможность процессу запускать другую программу, при этом соответствующий этой программе исполняемый файл будет располагаться в пространстве памяти процесса. Содержимое пользовательского контекста после вызова функции становится недоступным, за исключением передаваемых функции параметров, которые переписываются ядром из старого адресного пространства в новое.

Вызов exec возвращает 0 при успешном завершении и -1 при аварийном. В последнем случае управление возвращается в вызывающую программу.

В качестве примера использования этого вызова можно привести работу командного интерпретатора shell: при выполнении команды он сначала порождает свою копию с помощью вызова fork, а затем запускает соответствующую указанной команде программу системным вызовом exec.

## **Системные вызовы для работы с сигналами**

Рассмотрим наиболее часто используемые системные вызовы при работе с сигналами в ОС UNIX, описанные в библиотеке <signal.h>.

## KILL

Посылка всем или некоторым процессам любого сигнала:

```
int kill(pid, sig);
```

где sig - номер сигнала, pid - идентификатор процесса, определяющий группу родственных процессов, которым будет послан данный сигнал:

- если pid - положительное целое число, ядро посылает сигнал процессу с идентификатором pid.
- если значение pid равно 0, сигнал посылается всем процессам, входящим в одну группу с процессом, вызвавшим функцию kill.
- если значение pid равно -1, сигнал посылается всем процессам, у которых реальный код идентификации пользователя совпадает с тем, под которым выполняется процесс, вызвавший функцию kill. Если процесс, пославший сигнал, выполняется под кодом идентификации суперпользователя, сигнал рассылается всем процессам, кроме процессов с идентификаторами 0 и 1.
- если pid - отрицательное целое число, но не -1, сигнал посылается всем процессам, входящим в группу с номером, равным абсолютному значению pid.

Вызов kill возвращает 0 при успешном завершении и -1 при аварийном (например, спецификация несуществующего в ОС UNIX сигнала или несуществующего процесса).

Во всех случаях, если процесс, пославший сигнал, выполняется под кодом идентификации пользователя, не являющегося суперпользователем, или если коды идентификации пользователя (реальный и исполнительный) у этого процесса не совпадают с соответствующими кодами процесса, принимающего сигнал, kill завершается неудачно.

Посылка сигнала может сопровождать возникновение любого события. Сигналы SIGUSR1, SIGUSR2 и SIGKILL могут быть посланы только с помощью системного вызова kill.

## SIGNAL

Позволяет процессу самому определить свою реакцию на получение того или иного сигнала:

```
#include <signal.h>
```

```
void (*signal(signum, void (*handler)(int)))(int)
```

```
int signum;
```

```
void handler(int);
```

После определения реакции на сигнал `signal` при получении процессом этого сигнала будет автоматически вызываться функция `handler()`, которая, естественно, должна быть описана или объявлена прежде, чем будет осуществлен системный вызов `signal`.

При многократной обработке одного и того же сигнала, процесс должен каждый раз осуществлять системный вызов `signal` для установления требуемой реакции на данный сигнал. Использование констант `SIG_DFL` и `SIG_IGN` позволяет упростить реализацию двух часто встречающихся реакций процесса на сигнал:

`signal(SIGINT, SIG_IGN)` игнорирование сигнала;

`signal(SIGINT, SIG_DFL)` восстановление стандартной реакции на сигнал.

Аргументом функции-обработчика является целое число – номер обрабатываемого сигнала. Значение его устанавливается ядром.

## PAUSE

Приостанавливает функционирование процесса до получения им некоторого сигнала:

`void pause();`

Этот системный вызов не имеет параметров. Работа процесса возобновляется после получения им любого сигнала, кроме тех, которые игнорируются этим процессом.

## ALARM

Посылка процессу сигнала побудки `SIGALARM`:

`unsigned alarm(unsigned secs);`

Этим системным вызовом процесс информирует ядро ОС о том, что ядро должно послать этому процессу сигнал побудки через `secs` секунд. Вызов `alarm` возвращает число секунд, заданное при предыдущем осуществлении этого системного вызова.

Если `secs` равно 0, то специфицированная ранее посылка процессу сигнала `SIGALARM` будет отменена.

**GETPID** возвращает значение идентификатора текущего процесса. Системный вызов **getpid** (запрос идентификатора текущего процесса) имеет следующий формат:

`int getpid( )`

## GETGID, GETEGID

Системные вызовы **getgid**, **getegid** (запрос идентификатора группы) имеют форматы:

`int getgid( )`

**int getegid( )**

Вызов **getgid** возвращает реальный идентификатор группы текущего процесса, а вызов **getegid** - эффективный идентификатор группы. Реальный идентификатор группы определяется при регистрации в системе. Эффективный идентификатор определяет дополнительные права доступа при выполнении программы в режиме **set-GID** (установка идентификатора группы владельца выполняемого файла).

**GETUID, GETEUID**

Системные вызовы **getuid**, **geteuid** (запрос идентификатора пользователя) имеют следующие форматы:

**int getuid ( )**

**int geteuid ( )**

Системный вызов **getuid** возвращает реальный идентификатор пользователя для текущего процесса, а **geteuid** возвращает эффективный идентификатор пользователя. Реальный идентификатор пользователя совпадает с идентификатором, под которым пользователь зарегистрирован в системе. Эффективный идентификатор определяет права доступа пользователя в текущий момент. Такой механизм позволяет программе, работающей в режиме **set-UID** (установка прав доступа владельца объектного файла), определять, кто ее вызвал.

**SETGID**

Системный вызов **setgid** (установить реальный идентификатор группы процесса) имеет формат:

**setgid (rgid) int rgid;**

Системный вызов **setgid** устанавливает идентификатор группы текущего процесса в соответствии с значением параметра **rgid**. Изменять реальный идентификатор группы можно только в процессе, работающем в привилегированном режиме. В обычном режиме допускает только замена эффективного идентификатора на значение соответствующее реальному идентификатору группы процесса. При успешном завершении, возвращается значение 0, иначе - значение -1 и код ошибки в переменной **errno**. Код ошибки может принимать значение

[EPERM] (текущий процесс не является привилегированным, и были специфицированы изменения, отличные от замены эффективного идентификатора группы на реальный).

**SETUID**

Системный вызов **setuid** (установить реальный идентификаторы пользователя) имеет следующий формат:

**setuid (ruid) int ruid;**

Системный вызов **setuid** устанавливает текущему процессу реальный идентификатор пользователя в соответствии со значением параметра **ruid**. Изменять реальный идентификатор пользователя можно только в процессе, работающем в привилегированном режиме. В обычном режиме допускается только замена эффективного идентификатора на значение, соответствующее реальному идентификатору. При успешном завершении вызова, возвращается значение 0, иначе - значение -1 и код ошибки в переменной **errno**.

Код ошибки:

[**EPERM**] - текущий процесс не является привилегированным, и были специфицированы изменения, отличные от замены эффективного идентификатора на реальный.

Ниже приведен пример программы, распечатывающей на стандартный вывод информации о реальных и эффективных идентификаторах пользователя и группы для текущего выполняемого процесса, а также его идентификатор.

```
main ( )
{
    int    pid,uid, gid;
    int    euid, egid;
    pid = getpid();
    uid = getuid();
    gid = getgid();
    euid = geteuid();
    egid = getegid();
    printf("\tProcess: PID = %d\n", pid);
    printf("uid = %d gid = %d\n", uid, gid);
    printf("euid = %d egid = %d\n", euid, egid);
}
```

## NICE

Системный вызов **nice** (установить процессу приоритет) имеет следующий формат:

**nice (incr)**

Системный вызов **nice** устанавливается текущему процессу новый приоритет, равный текущему приоритету плюс значение аргумента **incr**. Значение аргумента **incr** должно лежать в определенном диапазоне, конкретные границы которого зависят от версии ОС UNIX.

UNIX 7. Диапазон значений **incr** лежит в пределах от -20 до 20. Отрицательное приращение может задавать только привилегированный пользователь.

UNIX System V. Диапазон значений **incr** лежит в пределах от 0 до 39. Приращения, выходящие за границы этого диапазона, уменьшаются до граничных значений. Чем выше значение аргумента **incr**, тем ниже устанавливаемый приоритет.

При успешном завершении вызова, возвращается новое значение приоритета минус 20, иначе - значение -1 и код ошибки в переменной **errno**.

Код ошибки:

[EPERM] Значение **incr** не лежит в пределах установленного диапазона и эффективный идентификатор владельца процесса не является идентификатором суперпользователя.

Приведенная ниже программа является исходным текстом команды **nice** UNIX версии System V (вызова какой-либо команды на выполнение с измененным приоритетом):

**nice** {приращение приоритета} команда

```
#include <stdio.h>
#include <ctype.h>
main (argc, argv) int argc; char *argv [];
{
    int nicarg = 10; /* приращение приоритета по умолчанию */
    extern errno;
    /* Разбираем аргументы команды */
    if(argc > 1 && argv[1][0] = '-') { /* если задано приращение приорите-
та */
        register char *p = argv[1];
        if(*++p != '-') --p;
        while(*++p)
            if(!isdigit(*p)) { /* указано не числовое значение */
                fprintf(stderr, "nice: argument must be numeric.\n");
                exit(2);
            }
        nicarg == atoi (&argv[1] [1]);
        /* в переменной nicarg записываем значение приращения приоритета
*/
        argc--;
        argv++;
    }
    if (argc < 2) { /* если не задана команда */
        fprintf(stderr, "nice: usage: nice [-num] command\n");
        exit(2);
    }
}
```



```

nice(nicarg); /* изменяем приоритет процесса */
execvp(argv [1], &argv[1]); /* выполняем команду */
fprintf(stderr. "%s: %d\n", argv[1], errno);
/* если ошибка при запуске команды, то печатаем переменную errno
*/
exit(2);
}

```

## ПРИМЕРЫ ПРОГРАММ

Пример 1. Порождение процессов.

Нижеприведенная программа в результате выполнения породит три процесса (процесс-предок 1 и процессы-потомки 2 и 3).

```

#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>

void main(void)
{
    int pid2, pid3, st; /* process 1 */
    printf(«Process 1, pid = %d:\n», getpid());
    pid2 = fork();
    if (pid2 == 0) /* process 2 */
    {
        printf(«Process 2, pid = %d:\n», getpid());
        pid3 = fork();
        if (pid3 == 0) /* process 3 */
        {
            printf(«Process 3, pid = %d:\n», getpid());
            sleep(2);
            printf(«Process 3: end\n»);
        } /* process 2 */
        if (pid3 < 0) printf(«Can't create process 3: error %d\n»,
pid3);
        wait(&st);
        printf(«Process 2: end\n»);
    }
    else /* process 1 */
    {
        if (pid2 < 0) printf(«Can't create process 2: error %d\n», pid2);
        wait(&st);
        printf(«Process 1: end\n»);
    }
    exit(0);
}

```

```
}
```

В соответствии с программой первоначально будет создан процесс1 (как потомок интерпретатора shell), он сообщит о начале своей работы и породит процесс2. После этого работа процесса1 приостановится и начнет выполняться процесс2 как более приоритетный. Он также сообщит о начале своей работы и породит процесс3. Далее начнет выполняться процесс3, он сообщит о начале работы и «заснет». После этого возобновит свое выполнение либо процесс1, либо процесс2 в зависимости от величин приоритетов и от того, насколько процессор загружен другими процессами. Так как ни один из процессов не выполняет никакой работы, они, вероятнее всего, успеют завершиться до возобновления процесса3, который в этом случае завершится последним.

#### Пример 2. Синхронизация работы процессов.

Приведенная в данном разделе программа в результате выполнения породит три процесса (процесс-предок 0 и процессы-потомки 1 и 2). Процессы 1 и 2 будут обмениваться сигналами и выдавать соответствующие сообщения на экран, а процесс 0 через определенное количество секунд отправит процессам 1 и 2 сигнал завершения и сам прекратит свое функционирование.

```
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

#define TIMEOUT 10

extern int f1(int), f2(int), f3(int);
int pid0, pid1, pid2;

void main(void)
{
    setpgrp();
    pid0 = getpid();
    pid1 = fork();
    if (pid1 == 0) /* process 1 */
    {
        signal(SIGUSR1, f1);
        pid1 = getpid();
        pid2 = fork();
        if (pid2 < 0 ) puts(«Fork error»);
```

```

        if (pid2 > 0) for(;;);
        else /* process 2 */
        {
            signal(SIGUSR2, f2);
            pid2 = getpid();
            kill(pid1, SIGUSR1);
            for (;;)
        }
    }
    else /* process 0 */
    {
        signal(SIGALRM, f3);
        alarm(TIMEOUT);
        pause();
    }
    exit(0);
}

int f1(int signum)
{
    signal(SIGUSR1, f1);
    printf(«Process 1 (%d) has got a signal from process 2
    (%d)\n», pid1, pid2);
    sleep(1);
    kill(pid2, SIGUSR2);
    return 0;
}

int f2(int signum)
{
    signal(SIGUSR2, f2);
    printf(«Process 2 (%d) has got a signal from process 1
    (%d)\n», pid2, pid1);
    sleep(1);
    kill(pid1, SIGUSR1);
    return 0;
}

int f3(int signum)
{
    printf(«End of job - %d\n», pid0);
    kill(0, SIGKILL);
    return 0;
}

```

### Задания на лабораторную работу

1. Процесс 1 порождает потомка 2, который в свою очередь порождает потомка 3. С помощью сигналов добиться того, чтобы эти процессы заканчивались в порядке 1, 2, 3.
2. Процесс 1 порождает потомка 2. Оба процесса после этого открывают один и тот же файл и пишут в него по очереди по N байт. Организовать M циклов записи с помощью сигналов.
3. Процесс 1 открывает файл и после этого порождает потомка 2. Процесс 2 начинает запись в файл после получения сигнала SIG1 от процесса 1 и прекращает ее после получения от процесса 1 сигнала SIG2, который посылается через N секунд после SIG1. Затем процесс 1 читает данные из файла и выводит их на экран.
4. Процесс 1 открывает файл и после этого порождает потомка 2. Один процесс пишет в файл один байт, посылает другому процессу сигнал, другой читает из файла один байт, выводит прочитанное на экран и посылает сигнал первому процессу. Организовать N циклов запись/чтение.
5. Процесс 1 открывает файл и порождает потомков 2 и 3. Потомки после сигнала от предка пишут в файл по N байт, посылают сигналы процессу 1 и завершают работу. После этого процесс 1 считывает данные из файла и выводит на экран.
6. Процесс 1 открывает файл и после этого порождает потомка 2, который в свою очередь порождает потомка 3. Процесс 2 пишет N байт в общий файл, посылает сигнал процессу 3, который тоже пишет N байт в файл и посылает сигнал процессу 1, который считывает данные из файла и выводит их на экран.
7. Процесс 1 открывает файл и порождает потомка 2, после этого пишет в файл N байт и посылает сигнал процессу 2. Процесс 2 пишет N байт в файл, посылает сигнал процессу 1 и завершается. Процесс 1, получив сигнал, считывает данные из файла, выводит их на экран и завершается.
8. Процесс 1 порождает потомка 2. Оба процесса открывают один и тот же файл и записывают в него в цикле по N байт. С помощью сигналов организовать очередность записи.
9. Процесс 1 открывает файл и порождает потомков 2 и 3. Потомки пишут в файл по очереди по N байт (M циклов записи, организовать с помощью сигналов) и завершаются. Последний из них посылает сигнал процессу 1, который читает данные и выводит их на экран.
10. Процесс 1 открывает файл, порождает потомка 2, пишет в файл N байт и посылает сигнал SIG1 процессу 2. Процесс 2 по сигналу SIG1 читает данные, выводит их на экран и завершается. Если сигнал от процесса 1

не поступит в течение  $M$  секунд, процесс 2 начинает считывать данные по сигналу SIGALRM.

11. Процесс 1 открывает файл и порождает потомка 2, который в свою очередь порождает потомка 3. Процессы 2 и 3 пишут в общий файл, причем процесс 3 не может начать писать раньше, чем процесс 2. Организовать очередность с помощью сигналов. Как только размер файла превысит  $N$  байт, процесс 1 посылает потомкам сигнал SIG2 о завершении работы, считывает данные из файла и выводит их на экран.

12. Процесс 1 открывает файл и порождает потомка 2. Процесс 1 с интервалом в 1 секунду (через alarm) посылает  $M$  сигналов SIG1 процессу 2, который по каждому сигналу пишет в общий файл по  $N$  чисел. Потом процесс 1 посылает процессу 2 сигнал SIG2, процесс 2 завершается. Процесс 1 считывает данные из файла и выводит их на экран.

13. Процесс 1 открывает файл и порождает потомков 2 и 3. Процесс 1 с интервалом в 1 секунду (через alarm) по очереди посылает  $N$  сигналов SIG1 процессам 2 и 3, которые по каждому сигналу пишут в общий файл по  $M$  чисел. Потом процесс 1 посылает потомкам сигнал SIG2, процессы 2 и 3 завершаются. Процесс 1 считывает данные из файла и выводит их на экран.

14. Процесс 1 открывает два файла и порождает потомков 2 и 3. Процессы 2 и 3 с интервалом в 1 секунду (через alarm) посылают по  $N$  сигналов процессу 1, который по каждому сигналу пишет в соответствующий файл по  $M$  чисел. Потом процессы 2 и 3 считывают данные из файлов, выводят их на экран и завершаются. Процесс 1 завершается последним.

15. Процесс 1 открывает два файла и порождает потомков 2 и 3. Процессы 2 и 3 посылают по одному сигналу процессу 1, который по каждому сигналу пишет в соответствующий файл  $M$  чисел. Процессы 2 и 3 считывают данные из файлов и выводят их на экран. С помощью сигналов организовать непересекающийся вывод данных.

16. Процесс 1 открывает файл и порождает потомка 2. Процесс 2 по сигналу SIG1 от процесса 1 начинает писать в файл, по сигналу SIG2 прекращает запись, а потом по сигналу SIG1 завершается. Сигнал SIG1 поступает с задержкой в 1 секунду относительно первого сигнала SIG2.

17. Процесс 1 открывает файл и порождает потомка 2. Процесс 1 читает данные из файла, процесс 2 пишет данные в файл следующим образом: по сигналу SIG1 от процесса 1 процесс 2 записывает в файл  $N$  чисел и посылает процессу 1 сигнал окончания записи; процесс 1 по этому сигналу считывает данные и посылает процессу 2 сигнал SIG1. Процесс 2 всегда пишет данные в начало файла. Организовать  $M$  циклов записи/чтения. Прочитанные данные выводятся на экран.

18. Процесс 1 открывает существующий файл и порождает потомка 2. Процесс 1 считывает  $N$  байт из файла, выводит их на экран и посылает

сигнал SIG1 процессу 2. Процесс 2 также считывает N байт из файла, выводит их на экран и посылает сигнал SIG1 процессу 1. Если одному из процессов встретился конец файла, то он посылает другому процессу сигнал SIG2 и они оба завершаются.

19.Процесс 1 открывает файл и порождает потомка 2. Оба процесса пишут в него по очереди по N чисел. Организовать M циклов записи с помощью сигналов.

20.Процесс 1 порождает потомка 2. Процесс 1 пишет в общий файл число 1, процесс 2 - число 2. Используя сигналы, обеспечить следующее содержимое файла:

- а) 1 2 1 2 1 2 1 2
- б) 1 1 2 2 1 1 2 2
- в) 1 1 2 1 1 2 1 1 2
- г) 2 1 1 2 1 1 2 1 1
- д) 1 2 2 1 2 2 1 2 2

### Контрольные вопросы

1. Каким образом может быть порожден новый процесс? Какова структура нового процесса?
2. Если процесс-предок открывает файл, а затем порождает процесс-потомок, а тот, в свою очередь, изменяет положение указателя чтения-записи файла, то изменится ли положение указателя чтения-записи файла процесса-отца?
3. Что произойдет, если процесс-потомок завершится раньше, чем процесс-предок осуществит системный вызов `wait()`?
4. Могут ли родственные процессы разделять общую память?
5. Каков алгоритм системного вызова `fork()`?
6. Какова структура таблиц открытых файлов, файлов и описателей файлов после создания процесса?
7. Каков алгоритм системного вызова `exit()`?
8. Каков алгоритм системного вызова `wait()`?
9. В чем разница между различными формами системных вызовов типа `exec()`?
10. Для чего используются сигналы в ОС UNIX?
11. Какие виды сигналов существуют в ОС UNIX?
12. Для чего используются каналы?
13. Какие требования предъявляются к процессам, чтобы они могли осуществлять обмен данными посредством каналов?
14. Каков максимальный размер программного канала и почему?

## Литература

1. Дансмур М., Дейвис Г. Операционная система UNIX и программирование на языке Си: Пер. с англ.- М.: "Радио и связь", 1989.-192 с.
2. Забродин Л.Д. UNIX. Введение в командный интерфейс. -М.: "ДИАЛОГ-МИФИ", 1994.-144 с.
3. Керниган Б.В., Пайк Р. UNIX - универсальная среда программирования: Пер. с англ.- М.: Финансы и статистика, 1992.-304 с.
4. Робачевский А.М. Операционная система UNIX. - СПб.: BHV - Санкт-Петербург, 1997. - 528 с.
5. Т. Чан Системное программирование на C++ для UNIX. /Пер. с англ. -К.: Издательская группа BHV, 1997. - 592 с.



Министерство образования и науки Российской Федерации

Федеральное агентство по образованию  
Государственное образовательное учреждение  
высшего профессионального образования  
«Комсомольский-на-Амуре Государственный технический универси-  
тет»  
Кафедра «Электропривод и автоматизация промышленных устано-  
вок»

**Разделяемая память и семафоры в ОС UNIX**  
Методические указания к лабораторной работе по курсу  
«СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ» для студентов направ-  
ления 220200.62 «Управление в технических системах для всех форм обу-  
чения

Комсомольск-на-Амуре 2011

УДК 004.4:004.31-181.4;004.318-181.4:004.4

Разделяемая память и семафоры в ОС UNIX: методические указания к лабораторной работе по курсам: «Системное программное обеспечение» /сост.: Черный С.П., Петренко Е.Д., Мешков А.С., – Комсомольск-на-Амуре : ГОУВПО «КНАГТУ», 2011. – 29 с.

Методические указания посвящены ознакомлению с семафорами как средством синхронизации работы параллельных процессов ОС UNIX, с обменом данными между процессами через разделяемую память; приобретение знаний и навыков написания программ работы с конкурирующими процессами.

Предназначены для студентов специальности 220201 «Управление и информатика в технических системах для всех форм обучения.

Печатается по постановлению редакционно-издательского совета ГОУВПО «Комсомольский-на-Амуре государственный технический университет».

Согласованно с патентно-информационным отделом.

Рецензент Соловьев В.А.

### **Цель работы:**

Состоит в изучении механизма взаимодействия процессов на основе разделяемой памяти, а также средств их синхронизации с использованием семафоров.

### **Теоретические сведения**

#### **Общие положения**

#### **Взаимодействие процессов в версии V системы UNIX**

Пакет IPC (Interprocess communication) в версии V системы UNIX включает в себя три механизма. Механизм сообщений дает процессам возможность посылать другим процессам потоки сформатированных данных, механизм разделения памяти позволяет процессам совместно использовать отдельные части виртуального адресного пространства, а семафоры - синхронизировать свое выполнение с выполнением параллельных процессов. Несмотря на то, что они реализуются в виде отдельных блоков, им присущи общие свойства.

С каждым механизмом связана таблица, в записях которой описываются все его детали.

В каждой записи содержится числовой ключ (key), который представляет собой идентификатор записи, выбранный пользователем.

В каждом механизме имеется системная функция типа “get”, используемая для создания новой или поиска существующей записи; параметрами функции являются идентификатор записи и различные флаги (flag). Ядро ведет поиск записи по ее идентификатору в соответствующей таблице. Процессы могут с помощью флага IPC\_PRIVATE гарантировать получение еще неиспользуемой записи. С помощью флага IPC\_CREAT они могут создать новую запись, если записи с указанным идентификатором нет, а если еще к тому же установить флаг IPC\_EXCL, можно получить уведомление об ошибке в том случае, если запись с таким идентификатором существует. Функция возвращает некий выбранный ядром дескриптор, предназначенный для последующего использования в других системных функциях, таким образом, она работает аналогично системным функциям creat и open.

В каждом механизме ядро использует следующую формулу для поиска по дескриптору указателя на запись в таблице структур данных: “указатель = <значение дескриптора> mod <число записей в таблице>”. Если, например, таблица записей разделяемой памяти состоит из 100 записей, дескрипторы, связанные с записью номер 1, имеют значения, равные 1, 101, 201 и т.д. Когда процесс удаляет запись, ядро увеличивает значение связанного с ней дескриптора на число записей в таблице: полученный де-

скриптор станет новым дескриптором этой записи, когда к ней вновь будет произведено обращение при помощи функции типа “get”. Процессы, которые будут пытаться обратиться к записи по ее старому дескриптору, потерпят неудачу. Обратимся вновь к предыдущему примеру. Если с записью 1 связан дескриптор, имеющий значение 201, при его удалении ядро назначит записи новый дескриптор, имеющий значение 301. Процессы, пытающиеся обратиться к дескриптору 201, получают ошибку, поскольку этого дескриптора больше нет. В конечном итоге ядро произведет перенумерацию дескрипторов, но пока это произойдет, может пройти значительный промежуток времени.

Каждая запись имеет некую структуру данных, описывающую права доступа к ней и включающую в себя пользовательский и групповой коды идентификации, которые имеет процесс, создавший запись, а также пользовательский и групповой коды идентификации, установленные системной функцией типа “control” (об этом ниже), и двоичные коды разрешений чтения-записи-исполнения для владельца, группы и прочих пользователей, по аналогии с установкой прав доступа к файлам.

В каждой записи имеется другая информация, описывающая состояние записи, в частности, идентификатор последнего из процессов, внесших изменения в запись (посылка сообщения, прием сообщения, подключение разделяемой памяти и т.д.), и время последнего обращения или корректировки.

В каждом механизме имеется системная функция типа “control”, запрашивающая информацию о состоянии записи, изменяющая эту информацию или удаляющая запись из системы. Когда процесс запрашивает информацию о состоянии записи, ядро проверяет, имеет ли процесс разрешение на чтение записи, после чего копирует данные из записи таблицы по адресу, указанному пользователем. При установке значений принадлежащих записи параметров ядро проверяет, совпадают ли между собой пользовательский код идентификации процесса и идентификатор пользователя (или создателя), указанный в записи, не запущен ли процесс под управлением суперпользователя; одного разрешения на запись недостаточно для установки параметров. Ядро копирует сообщенную пользователем информацию в запись таблицы, устанавливая значения пользовательского и группового кодов идентификации, режимы доступа и другие параметры (в зависимости от типа механизма). Ядро не изменяет значения полей, описывающих пользовательский и групповой коды идентификации создателя записи, поэтому пользователь, создавший запись, сохраняет управляющие права на нее. Пользователь может удалить запись, либо если он является суперпользователем, либо если идентификатор процесса совпадает с любым из идентификаторов, указанных в структуре записи. Ядро увеличивает номер дескриптора, чтобы при следующем назначении записи ей был при-

своем новый дескриптор. Следовательно, как уже ранее говорилось, если процесс попытается обратиться к записи по старому дескриптору, вызванная им функция получит отказ. Для использования механизмов IPC необходимо подключить к программе файл `<sys/ipc.h>`.

### **Использование разделяемой памяти**

Процессы могут взаимодействовать друг с другом непосредственно путем разделения (совместного использования) участков виртуального адресного пространства и обмена данными через разделяемую память. Процессы ведут чтение и запись данных в области разделяемой памяти, используя для этого те же самые машинные команды, что и при работе с обычной памятью.

После создания области разделяемой памяти и присоединения ее к виртуальному адресному пространству процесса эта область становится доступна так же, как любой участок виртуальной памяти; для доступа к находящимся в ней данным не нужны обращения к каким-то дополнительным системным функциям.

Механизм разделения памяти имеет много общего с механизмом функционирования файловой системы. Но в отличие от файлов ядро не располагает сведениями о том, какие процессы могут использовать механизм разделяемой памяти, а, следовательно, оно не может автоматически очищать неиспользуемые структуры механизма взаимодействия процессов, поскольку ядру неизвестно, какие из этих структур больше не нужны. Таким образом, завершившиеся вследствие возникновения ошибки процессы могут оставить после себя ненужные и неиспользуемые структуры, перегружающие и засоряющие систему. Для того, чтобы избежать подобных ситуаций, необходимо с большой осторожностью пользоваться этим механизмом и обязательно удалять разделяемую память после использования.

### **Семафоры**

В настоящее время все большее значение придается крупным вычислительным системам, таким как многопроцессорные вычислительные комплексы и сети ЭВМ. Основным средством увеличения мощностей вычислительных машин является повышение в них уровня параллелизма. Очевидно, что параллельность в аппаратуре отражается и на программном обеспечении. В связи с этим значительно возрастает интерес к параллельным процессам и проблемам их синхронизации.

На сегодняшний день предложено большое количество различных систем синхронизации процессов. К ним относятся:

- блокировка памяти;
- семафоры;

- критические области;
- условные критические области;
- мониторы;
- исключаяющие области и т.д..

Один из способов синхронизации параллельных процессов – семафоры. Дейкстры, реализованные в ОС UNIX.

### **Синхронизация процессов**

Определим понятие процесса как задания, выполняемого в операционной системе. Поскольку задания в многопользовательской системе должны выполняться независимо и могут выполняться параллельно, то и представляющие их процессы должны быть независимыми и параллельно выполняемыми.

Процессам для работы часто требуются различные устройства и вспомогательные программы, которые можно назвать соответственно аппаратными и программными ресурсами. Если мы хотим эффективно использовать эти ресурсы, то те и другие должны совместно использоваться несколькими процессами. Такие совместно используемые ресурсы называются разделяемыми ресурсами.

Если по условиям работы требуется, чтобы разделяемые ресурсы одновременно были доступны только одному процессу, то такие ресурсы называются критическими.

Под независимостью процессов понимается, что кроме (достаточно редких) моментов явной связи, процессы рассматриваются как совершенно независимые друг от друга. Но на самом деле они не являются вполне независимыми, так как они могут использовать в процессе своего выполнения одни и те же ресурсы. Процесс упорядочения общения между конкурирующими процессами называется синхронизацией. Синхронизация задается с помощью синхронизирующих правил. Реализация таких правил осуществляется с помощью средств синхронизации.

Элементарные приемы синхронизации, такие как использование общих переменных, имеют ряд недостатков, которые иногда приводят к невозможности получения правильных решений. Поэтому возникла необходимость в создании специальных синхронизирующих примитивов.

Такие примитивы под названием P и V операции были предложены Дейкстрой в 1968 году. Эти операции могут выполняться только над специальными переменными, называемыми семафорами или семафорными переменными. Семафоры являются целыми величинами и первоначально были определены как принимающие только неотрицательные значения. Кроме того, если их использовать для решения задач взаимного исключения, то область их значений может быть ограничена лишь “0” или “1”. Однако в дальнейшем была показана важная область применения семафоров,

принимающих любые целые положительные значения. Такие семафоры получили название “общих семафоров” в отличие от “двоичных семафоров”, используемых в задачах взаимного исключения. Р и V операции являются единственными операциями, выполняемыми над семафорами. Иногда они называются семафорными операциями.

Дадим определение Р и V операций в том виде, в котором они были предложены Дейкстрой.

**V - операция (V(S)):**

операция с одним аргументом, который должен быть семафором.

Эта операция увеличивает значение аргумента на 1.

**P - операция (P(S)):**

операция с одним аргументом, который должен быть семафором. Ее назначение - уменьшить величину аргумента на 1, если только результирующее значение не становится отрицательным.

Завершение Р-операции, т.е. решение о том, что настоящий момент является подходящим для выполнения уменьшения и последующее собственно уменьшение значения аргумента, должно рассматриваться как неделимая операция.

Эти определения справедливы как для общих, так для двоичных семафоров.

### **Реализация семафоров**

Системные вызовы для работы с семафорами содержатся в пакете IPC (подключаемый файл описаний - <sys/ipc.h>). Эти вызовы обеспечивают синхронизацию выполнения параллельных процессов, производя набор действий только над группой семафоров (средствами низкого уровня).

UNIX поддерживает числовые семафоры (как расширение двоичных семафоров). Семафоры UNIX носят не обязательный, а уведомительный характер. Это означает, что связь между семафором и тем ресурсом (ресурсами), доступ к которому разграничивает данный семафор, является чисто логической. Если при обращении к этому ресурсу процесс не запросит доступ к нему через семафор, никто не помешает процессу получить этот доступ (при наличии соответствующих прав). Таким образом, процессы должны заранее договариваться об использовании семафоров.

Каждый семафор в системе UNIX представляет собой набор значений (вектор семафоров). Связанные с семафорами системные функции являются обобщением операций Р и V семафоров Дейкстры, в них допускается одновременное выполнение нескольких операций (над семафорами, принадлежащими одному вектору, так называемые векторные операции). Ядро выполняет операции комплексно; ни один из посторонних процессов не сможет переустанавливать значения семафоров, пока все операции не

будут выполнены. Если ядро по каким-либо причинам не может выполнить все операции, оно не выполняет ни одной; процесс приостанавливает свою работу до тех пор, пока эта возможность не будет предоставлена. (Подробнее о порядке операций над семафорами см. п. 2. “Системные вызовы”).

Семафор в System V состоит из следующих элементов:

значение семафора,  
идентификатор последнего из процессов, работавших с семафором,  
количество процессов, ожидающих увеличения значения семафора,  
количество процессов, ожидающих момента, когда значение семафора станет равным 0.

Для создания набора семафоров и получения доступа к ним используется системная функция `semget`, для выполнения различных управляющих операций над набором - функция `semctl`, для работы со значениями семафоров - функция `semop`.

### **Общие замечания**

Механизм функционирования файловой системы и механизмы взаимодействия процессов имеют ряд общих черт. Системные функции типа “get” похожи на функции `creat` и `open`, функции типа “control” (`ctl`) предоставляют возможность удалять дескрипторы из системы, чем похожи на функцию `unlink`. Тем не менее, в механизмах взаимодействия процессов отсутствуют операции, аналогичные операциям, выполняемым системной функцией `close`. Следовательно, ядро не располагает сведениями о том, какие процессы используют механизм IPC, и, действительно, процессы могут прибегать к услугам этого механизма, если правильно “угадывают” соответствующий идентификатор и если у них имеются необходимые права доступа, даже если они не выполнили предварительно функцию типа “get”.

Выше уже говорилось, что ядро не может автоматически очищать неиспользуемые структуры механизма взаимодействия процессов, поскольку ядру неизвестно, какие из этих структур больше не нужны. Таким образом, завершившиеся вследствие возникновения ошибки процессы могут оставить после себя ненужные и неиспользуемые структуры, перегружающие и засоряющие систему. Несмотря на то, что в структурах механизма взаимодействия после завершения существования процесса ядро может сохранить информацию о состоянии и данные, лучше для этих целей использовать файлы.

Вместо традиционных, получивших широкое распространение файлов механизмы взаимодействия процессов используют новое пространство имен, состоящее из ключей (`keys`). Расширить семантику ключей на всю сеть довольно трудно, поскольку на разных машинах ключи могут описывать различные объекты. Короче говоря, ключи в основном предназначены



для использования в однопользовательских системах. Имена файлов в большей степени подходят для распределенных систем. Использование ключей вместо имен файлов также свидетельствует о том, что средства взаимодействия процессов являются “вещью в себе”, полезной в специальных приложениях, но не имеющей тех возможностей, которыми обладают, например, каналы и файлы.

Большая часть функциональных возможностей, предоставляемых данными средствами, может быть реализована с помощью других системных средств, поэтому включать их в состав ядра вряд ли следовало бы. Тем не менее, их использование в составе пакетов прикладных программ тесного взаимодействия дает лучшие результаты по сравнению со стандартными файловыми средствами.

## **СИСТЕМНЫЕ ВЫЗОВЫ**

### **Системные вызовы для работы с разделяемой памятью**

Системные вызовы для работы с разделяемой памятью в ОС UNIX описаны в библиотеке `<sys/shm.h>`.

Функция `shmget` создает новую область разделяемой памяти или возвращает адрес уже существующей области, функция `shmat` логически присоединяет область к виртуальному адресному пространству процесса, функция `shmdt` отсоединяет ее, а функция `shmctl` позволяет получать информацию о состоянии разделяемой памяти и производить над ней операции.

### **SHMGET**

Создание области разделяемой памяти или получение номера дескриптора существующей области:

```
int shmget(key_t key, int size, int flag);
id = shmget(key, size, flag);
```

где `id` - идентификатор области разделяемой памяти, `key` - номер области, `size` - объем области в байтах, `flag` - параметры создания и права доступа.

Ядро использует `key` для ведения поиска в таблице разделяемой памяти: если подходящая запись обнаружена и если разрешение на доступ имеется, ядро возвращает вызывающему процессу указанный в записи дескриптор. Если запись не найдена и пользователь установил флаг `IPC_CREAT`, указывающий на необходимость создания новой области, ядро проверяет нахождение размера области в установленных системой пределах и выделяет область.

Ядро записывает установки прав доступа, размер области и указатель на соответствующую запись таблицы областей в таблицу разделяемой па-

мяти (Рис.1) и устанавливает флаг, свидетельствующий о том, что с областью не связана отдельная память.

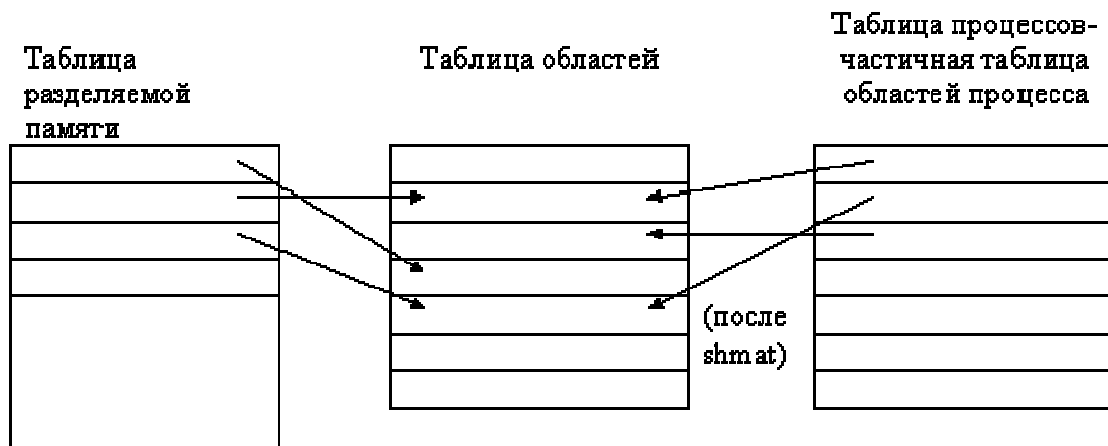


Рисунок - 1. Структуры данных, используемые при разделении памяти.

Области выделяется память (таблицы страниц и т.п.) только тогда, когда процесс присоединяет область к своему адресному пространству. Ядро устанавливает также флаг, говорящий о том, что по завершении последнего связанного с областью процесса область не должна освобождаться. Таким образом, данные в разделяемой памяти остаются в сохранности, даже если она не принадлежит ни одному из процессов (как часть виртуального адресного пространства последнего).

## SHMAT

Присоединяет область разделяемой памяти к виртуальному адресному пространству процесса:

```
void *shmat(int id, void *addr, int flag);
virtaddr = shmat(id, addr, flag);
```

Значение `id`, возвращаемое функцией `shmget`, идентифицирует область разделяемой памяти, `addr` является виртуальным адресом, по которому пользователь хочет подключить область, а с помощью флагов (`flag`) можно указать, предназначена ли область только для чтения и нужно ли ядру округлять значение указанного пользователем адреса. Возвращаемое функцией значение, `virtaddr`, представляет собой виртуальный адрес, по которому ядро произвело подключение области и который не всегда совпадает с адресом, указанным пользователем. В начале выполнения системной функции `shmat` ядро проверяет наличие у процесса необходимых прав доступа к области. Оно исследует указанный пользователем адрес; если он равен 0, ядро выбирает виртуальный адрес по своему усмотрению. Область разделяемой памяти не должна пересекаться в виртуальном адресном пространстве процесса с другими областями; следовательно, ее выбор должен

производиться разумно и осторожно. Так, например, процесс может увеличить размер принадлежащей ему области данных с помощью системного вызова `brk`, и новая область данных будет содержать адреса, смежные с прежней областью; поэтому ядру не следует присоединять область разделяемой памяти слишком близко к области данных процесса. Так же не следует размещать область разделяемой памяти вблизи от вершины стека, чтобы стек при своем последующем увеличении не залезал за ее пределы. Если, например, стек растет в направлении увеличения адресов, лучше всего разместить область разделяемой памяти непосредственно перед началом области стека. Ядро проверяет возможность размещения области разделяемой памяти в адресном пространстве процесса и присоединяет ее, если это возможно.

Если вызывающий процесс является первым процессом, который присоединяет область, ядро выделяет для области все необходимые таблицы, записывает время присоединения в соответствующее поле таблицы разделяемой памяти и возвращает процессу виртуальный адрес, по которому область была им подключена фактически.

### **SHMDT**

Отсоединение области разделяемой памяти от виртуального адресного пространства процесса:

```
int shmdt(void *addr);
```

где `addr` - виртуальный адрес, возвращенный функцией `shmat`. Процесс использует виртуальный адрес разделяемой памяти, а не ее идентификатор, поскольку этот идентификатор может быть удален из системы. Ядро производит поиск области по указанному адресу и отсоединяет ее от адресного пространства процесса. Поскольку в таблицах областей отсутствуют обратные указатели на таблицу разделяемой памяти, ядру приходится просматривать таблицу разделяемой памяти в поисках записи, указывающей на данную область, и записывать в соответствующее поле время последнего отключения области.

Отсоединение области от виртуального адресного пространства процесса не означает удаления области: сведения о ней остаются в таблице разделяемой памяти, данные, содержащиеся в ней, также сохраняются. Очищения таблиц и освобождения памяти можно добиться с помощью соответствующего флага в операции `shmctl`.

### **SHMCTL**

Получение информации о состоянии области разделяемой памяти и установка параметров для нее:

```
int shmctl(int id, int cmd, struct shmid_ds *buf);
```

Значение `id` (возвращаемое функцией `shmget`) идентифицирует запись таблицы разделяемой памяти, `cmd` определяет тип операции, а `buf` является адресом пользовательской структуры, хранящей информацию о состоянии области. Типы операций описываются списком определений в файле `"sys/ipc.h"`:

```
#define IPC_RMID 10 /* удалить идентификатор (область) */
#define IPC_SET 11 /* установить параметры */
#define IPC_STAT 12 /* получить параметры */
```

С помощью команды (флага) `IPC_RMID` можно удалить область `id`. Удаляя область разделяемой памяти, ядро освобождает соответствующую ей запись в таблице разделяемой памяти и просматривает таблицу областей: если область не была присоединена ни к одному из процессов, ядро освобождает запись таблицы и все выделенные области ресурсы. Если же область по-прежнему подключена к каким-то процессам (значение счетчика ссылок на нее больше 0), ядро только сбрасывает флаг, говорящий о том, что по завершении последнего связанного с нею процесса область не должна освобождаться. Процессы, уже использующие область разделяемой памяти, продолжают работать с ней, новые же процессы не могут присоединить ее. Когда все процессы отключат область, ядро освободит ее. Это похоже на то, как в файловой системе после разрыва связи с файлом процесс может вновь открыть его и продолжать с ним работу.

### **Системные вызовы для работы с семафорами**

Системные вызовы для работы с семафорами в ОС UNIX описаны в библиотеке `<sys/sem.h>`.

### **SEMGET**

Создание набора семафоров и получение доступа к ним:

```
int semget(key_t key, int count, int semflg);
semid = semget(key, count, semflg);
```

где `key` - номер семафора, `count` - количество семафоров, `semflg` - параметры создания и права доступа. Ядро использует `key` для ведения поиска в таблице семафоров: если подходящая запись обнаружена и разрешение на доступ имеется, ядро возвращает вызывающему процессу указанный в записи дескриптор. Если запись не найдена, а пользователь установил флаг `IPC_CREAT` - создание нового семафора, - ядро проверяет возможность его создания и выделяет запись в таблице семафоров. Запись указывает на массив семафоров и содержит счетчик `count` (Рис.2). В записи также хранится количество семафоров в массиве, время последнего выполнения функций `semop` и `semctl`.

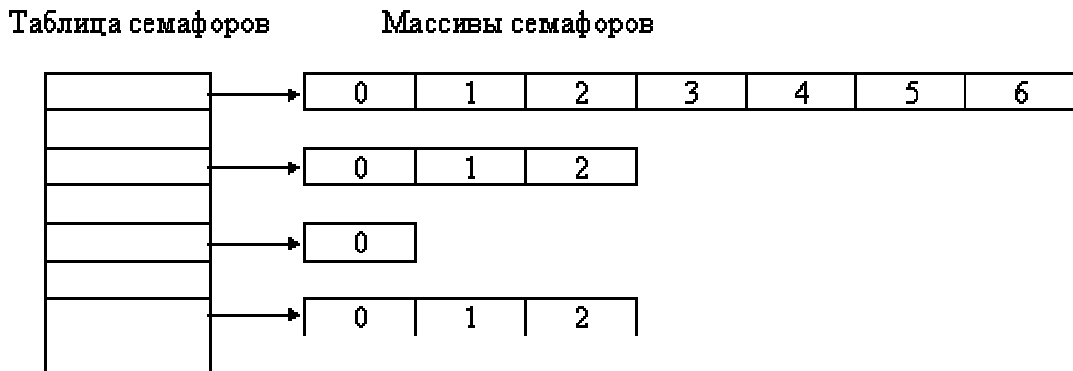


Рисунок - 2. Структуры данных, используемые в работе над семафорами.

## SEMOP

Установка или проверка значения семафора:

```
int semop(int semid, struct sembuf *oplist, unsigned nsops);
```

где `semid` - дескриптор, возвращаемый функцией `semget`, `oplist` - указатель на список операций, `nsops` - размер списка. Возвращаемое функцией значение является прежним значением семафора, над которым производилась операция. Каждый элемент списка операций имеет следующий формат (определение структуры `sembuf` в файле `sys/sem.h`):

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

где `short sem_num` - номер семафора, идентифицирующий элемент массива семафоров, над которым выполняется операция; `sem_op` - код операции; `sem_flg` - флаги операции. Ядро считывает список операций `oplist` из адресного пространства задачи и проверяет корректность номеров семафоров, а также наличие у процесса необходимых разрешений на чтение и корректировку семафоров. Если таких разрешений не имеется, системная функция завершается неудачно (`res = -1`). Если ядру приходится приостанавливать свою работу при обращении к списку операций, оно возвращает семафорам их прежние значения и находится в состоянии приостанова до наступления ожидаемого события, после чего системная функция запускается вновь. Поскольку ядро хранит коды операций над семафорами в глобальном списке, оно вновь считывает этот список из пространства задачи, когда перезапускает системную функцию. Таким образом, операции выполняются комплексно - или все за один сеанс, или ни одной.

Установка флага `IPC_NOWAIT` в функции `semop` имеет следующий смысл: если ядро попадает в такую ситуацию, когда процесс должен при-

остановить свое выполнение в ожидании увеличения значения семафора выше определенного уровня или, наоборот, снижения этого значения до 0, и если при этом флаг `IPC_NOWAIT` установлен, ядро выходит из функции с извещением об ошибке. (Таким образом, если не приостанавливать процесс в случае невозможности выполнения отдельной операции, можно реализовать условный тип семафора). Флаг `SEM_UNDO` позволяет избежать блокирования семафора процессом, который закончил свою работу прежде, чем освободил захваченный им семафор. Если процесс установил флаг `SEM_UNDO`, то при завершении этого процесса ядро даст обратный ход всем операциям, выполненным процессом. Для этого в распоряжении у ядра имеется таблица, в которой каждому процессу отведена отдельная запись. Запись содержит указатель на группу структур восстановления, по одной структуре на каждый используемый процессом семафор (Рис.3).

Каждая структура восстановления состоит из трех элементов - идентификатора семафора, его порядкового номера в наборе и установочного значения. Ядро выделяет структуры восстановления динамически, во время первого выполнения системной функции `semop` с установленным флагом `SEM_UNDO`. При последующих обращениях к функции с тем же флагом ядро просматривает структуры восстановления для процесса в поисках структуры с тем же самым идентификатором и порядковым номером семафора, что и в вызове функции. Если структура обнаружена, ядро вычитает значение произведенной над семафором операции из установочного значения. Таким образом, в структуре восстановления хранится результат вычитания суммы значений всех операций, произведенных над семафором, для которого установлен флаг `SEM_UNDO`.

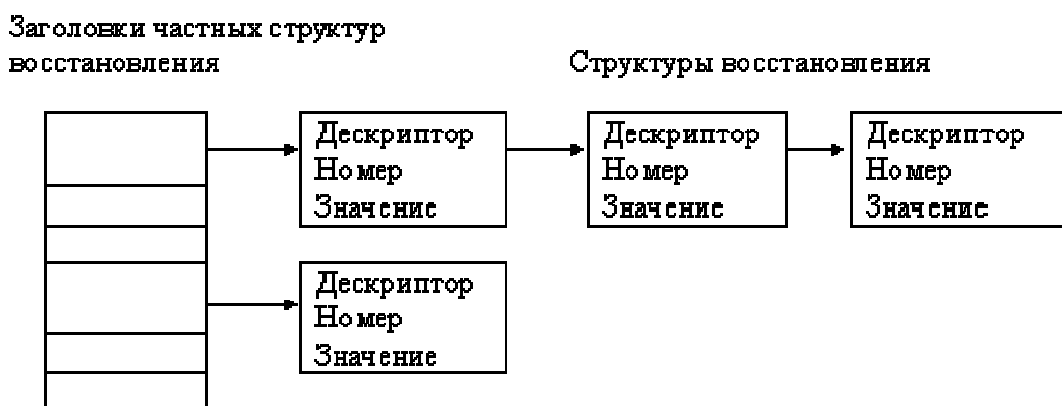


Рисунок - 3. Структуры восстановления семафоров

Если соответствующей структуры нет, ядро создает ее, сортируя при этом список структур по идентификаторам и номерам семафоров. Если установочное значение становится равным 0, ядро удаляет структуру из списка. Когда процесс завершается, ядро вызывает специальную процеду-

ру, которая просматривает все связанные с процессом структуры восстановления и выполняет над указанным семафором все обусловленные действия.

Ядро меняет значение семафора в зависимости от кода операции, указанного в вызове функции `semop`. Если код операции имеет положительное значение, ядро увеличивает значение семафора и выводит из состояния приостанова все процессы, ожидающие наступления этого события. Если код операции равен 0, ядро проверяет значение семафора: если оно равно 0, ядро переходит к выполнению других операций; в противном случае ядро увеличивает число приостановленных процессов, ожидающих, когда значение семафора станет нулевым, и “засыпает”.

Если код операции отрицателен и если его абсолютное значение не превышает значение семафора, ядро прибавляет код операции (отрицательное число) к значению семафора. Если результат равен 0, ядро выводит из состояния приостанова все процессы, ожидающие обнуления значения семафора. Если результат меньше абсолютного значения кода операции, ядро приостанавливает процесс до тех пор, пока значение семафора не увеличится. Если процесс приостанавливается посреди операции, он имеет приоритет, допускающий прерывания; следовательно, получив сигнал, он выходит из этого состояния.

## SEMCTL

Выполнение управляющих операций над набором семафоров:

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

Параметр `arg` объявлен как объединение типов данных:

```
union semunion
```

```
{ int val; // используется только для SETVAL
```

```
struct semid_ds *semstat; // для IPC_STAT и IPC_SET
```

```
unsigned short *array;
```

```
} arg;
```

Ядро интерпретирует параметр `arg` в зависимости от значения параметра `cmd`, который может принимать следующие значения:

**GETVAL** - вернуть значение того семафора, на который указывает параметр `num`.

**SETVAL** - установить значение семафора, на который указывает параметр `num`, равным значению `arg.val`.

**GETPID** - вернуть идентификатор процесса, выполнявшего последней функцией `semop` по отношению к тому семафору, на который указывает параметр `semnum`.

**GETNCNT** - вернуть число процессов, ожидающих того момента, когда значение семафора станет положительным.

GETZCNT - вернуть число процессов, ожидающих того момента, когда значение семафора станет нулевым.

GETALL - вернуть значения всех семафоров в массиве arg.array.

SETALL - установить значения всех семафоров в соответствии с содержимым массива arg.array.

IPC\_STAT - считать структуру заголовка семафора с идентификатором id в буфер arg.buf. Аргумент semnum игнорируется.

IPC\_SET - запись структуры семафора из буфера arg.buf.

IPC\_RMID - удалить семафоры, связанные с идентификатором id, из системы.

Если указана команда удаления IPC\_RMID, ядро ведет поиск всех процессов, содержащих структуры восстановления для данного семафора, и удаляет соответствующие структуры из системы. Затем ядро инициализирует используемые семафором структуры данных и выводит из состояния приостанова все процессы, ожидающие наступления некоторого связанного с семафором события: когда процессы возобновляют свое выполнение, они обнаруживают, что идентификатор семафора больше не является корректным, и возвращают вызывающей программе ошибку. Если возвращаемое функцией число равно 0, то функция завершилась успешно, иначе (возвращаемое значение равно -1) произошла ошибка. Код ошибки хранится в переменной errno.

## ПРИМЕРЫ ПРОГРАММ

Пример 1. Запись в область разделяемой памяти и чтение из нее.

Первая из программ описывает процесс, в котором создается область разделяемой памяти размером 128 Кбайт и производится запись и считывание данных из этой области.

В соответствии со второй программой другой процесс присоединяет ту же область (он получает только 64 Кбайта, таким образом, каждый процесс может использовать разный объем области разделяемой памяти); он ждет момента, когда первый процесс запишет в первое принадлежащее области слово любое отличное от нуля значение, и затем принимается считывать данные из области.

Первый процесс делает “паузу” (pause), предоставляя второму процессу возможность выполнения; когда первый процесс принимает сигнал, он удаляет область разделяемой памяти из системы.

```
/*
Запись в разделяемую память и чтение из нее
*/
#include <sys/types.h>
#include <sys/ipc.h>
```



```

#include <sys/shm.h>
#define SHMKEY 5
#define K 1024
int shmid;
main()
{ int i, *pint;
  char *addr;
  extern char *shmat();
  extern cleanup();
  /* определение реакции на все сигналы */
  for (i = 0; i < 20; i++) signal(i, cleanup);
  /* создание общедоступной разделяемой области памяти размером
128*K (или получение ее идентификатора, если она уже существует) */
  shmid = shmget(SHMKEY,128*K,0777|IPC_CREAT);
  addr = shmat(shmid,0,0);
  pint = (int *) addr;
  for (i = 0; i < 256; i++) *pint++ = i;
  pint = (int *) addr;
  *pint = 256;
  pint = (int *) addr;
  for (i = 0; i < 256; i++)
    printf("index %d\tvalue %d\n",i,*pint++);
  /* ожидание сигнала */
  pause();
}
/* удаление разделяемой памяти */
cleanup()
{
  shmctl(shmid,IPC_RMID,0);
  exit();
}
/*
Чтение из разделяемой памяти данных, записанных первым
процессом
*/

```

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHMKEY 75
#define K 1024
int shmid;
main()

```

```

{ int i, *pint;
  char *addr;
  extern char *shmat();
  shmid = shmget(SHMKEY, 64*K, 0777);
  addr = shmat(shmid, 0, 0);
  pint = (int *) addr;
  while (*pint == 0); /* ожидание начала записи */
  for (i = 0; i < 256, i++) printf("%d\n", *pint++);
}

```

Пример 2. Работа двух параллельных процессов в одном критическом интервале времени.

Для организации работы двух процессов в одном критическом интервале времени необходимо время работы одного процесса сделать недоступным для другого (т.е. второй процесс не может выполняться одновременно с первым). Для этого используем средство синхронизации - семафор. В данном случае нам потребуется один семафор. Опишем его с помощью системной функции ОС UNIX:

```
lsid = semget(75, 1, 0777 | IPC_CREAT);
```

где `lsid` - это идентификатор семафора; 75 - ключ пользовательского дескриптора (если он занят, система создаст свой); 1 - количество семафоров в массиве; `IPC_CREAT` - флаг для создания новой записи в таблице дескрипторов (описан с правами доступа 0777).

Для установки начального значения семафора используем структуру `sem`. В ней присваиваем значение:

```
sem.array[0] = 1;
```

то есть семафор открыт для пользования.

Завершающим шагом является инициализация массива (в данном случае массив состоит из одного элемента):

```
semctl(lsid, 1, SETALL, sem);
```

где `lsid` - идентификатор семафора (выделенная строка в дескрипторе); 1 - количество семафоров; `SETALL` - команда "установить все семафоры"; `sem` - указатель на структуру.

Устанавливаем флаг `SEM_UNDO` в структуре `sop` для работы с функцией `semop` (значение этого флага не меняется в процессе работы).

Далее в программе организуются два параллельных процесса (потомки "главной" программы) с помощью системной функции `fork()`. Один процесс-потомок записывает данные в разделяемую память, второй считывает эти данные. При этом процессы-потомки синхронизируют доступ к разделяемой памяти с помощью семафоров.

Функция `p()` описывается следующим образом:

```

int p(int sid)
{ sop.sem_num = 0; /* номер семафора */

```

```
sop.sem_op = -1;
semop(sid, &sop, 1);
}
```

В структуру `sop` заносится номер семафора, над которым будет произведена операция и значение самой операции (в данном случае это уменьшение значения на 1). Флаг был установлен заранее, поэтому функция в процессе всегда находится в ожидании свободного семафора. (Функция `v()` работает аналогично, но `sop.sem_op = 1`).

Результатом выполнения ниже приведенной программы является список сообщений от процессов, соответствующий последовательности работы процессов.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <unistd.h>
#include <errno.h>
int shmid, lsid, x;
struct sembuf sop;
union semun
{ int val;
  struct semid_ds *buf;
  ushort *array;
} sem;
int p(int sid)
{ sop.sem_num = 1;
  sop.sem_op = -1;
  semop(sid, &sop, 1);
}
int v(int sid)
{ sop.sem_num = 1;
  sop.sem_op = 1;
  semop(sid, &sop, 1);
}
main()
{ int j, i, id, id1, n;
  lsid = semget(75, 1, 0777 | IPC_CREAT);
  sem.array = (ushort*)malloc(sizeof(ushort));
  sem.array[0] = 1;
  sop.sem_flg = SEM_UNDO;
  semctl(lsid, 1, SETALL, sem);
```

```

printf(" n= ");
scanf("%d", &n);
id = fork();
if (id == 0) /* первый процесс */
{ for(i = 0; i < n; i++)
{ p(lsid);
puts("\n Работает процесс 1");
v(lsid);
}
exit(0);
}
id1 = fork();
if (id1 == 0) /* второй процесс */
{ for (j = 0; j < n; j++)
{ p(lsid);
puts("\n Работает процесс 2");
v(lsid);
}
exit(0);
}
wait(&x);
wait(&x);
exit(0);
}

```

Пример 3. Векторные операции над семафорами.

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define SEMKEY 75
int semid; /* идентификатор семафора */
unsigned int count; /* количество семафоров */
struct sembuf psembuf, vsembuf; /* операции типа P и V */
cleanup()
{ semctl(semid, 2, IPC_RMID, 0);
exit();
}
main(int argc, char *argv[])
{ int i, first, second;
short initarray[2], outarray[2];
if (argc == 1)
{
/* определение реакции на сигналы */

```

```

for (i = 0; i < 20; i++) signal(i, cleanup);
/* создание общедоступного семафора из двух элементов */
semid = semget(SEMKEY, 2, 0777|IPC_CREAT);
/* инициализация семафоров (оба открыты) */
initarray[0] = initarray[1] = 1;
semctl(semid, 2, SETALL, initarray);
semctl(semid, 2, GETALL, outarray);
printf("начальные значения семафоров %d %d\n",
outarray[0],outarray[1]);
pause(); /* приостанов до получения сигнала */
}
else
if (argv[1][0] == 'a')
{ first = 0;
second = 1;
}
else
{ first = 1;
second = 0;
}
/* получение доступа к ранее созданному семафору */
semid = semget(SEMKEY, 2, 0777);
/* определение операций P и V */
psembuf.sem_op = -1;
psembuf.sem_flg = SEM_UNDO;
vsembuf.sem_op = 1;
vsembuf.sem_flg = SEM_UNDO;
for (count = 0; ; count++)
{
/* закрыть первый семафор */
psembuf.sem_num = first;
semop(semid, &psembuf, 1);
/* закрыть второй семафор */
psembuf.sem_num = second;
semop(semid, &psembuf, 1);
printf("процесс %d счетчик %d\n", getpid(), count);
/* открыть второй семафор */
vsembuf.sem_num = second;
semop(semid, &vsembuf, 1);
/* открыть первый семафор */
vsembuf.sem_num = first;
semop(semid, &vsembuf, 1);

```

```

}
}

```

Предположим, что пользователь исполняет данную программу (под именем a.out) три раза в следующем порядке:

```

a.out &
a.out a &
a.out b &

```

Если программа вызывается без параметров, процесс создает набор семафоров из двух элементов и присваивает каждому семафору значение, равное 1. Затем процесс вызывает функцию pause() и приостанавливается для получения сигнала, после чего удаляет семафор из системы (cleanup).

При выполнении программы с параметром 'a' процесс (A) производит над семафорами в цикле четыре операции: он уменьшает на единицу значение семафора 0, то же самое делает с семафором 1, выполняет команду вывода на печать и вновь увеличивает значения семафоров 0 и 1. Если бы процесс попытался уменьшить значение семафора, равное 0, ему пришлось бы приостановиться, следовательно, семафор можно считать захваченным (недоступным для уменьшения). Поскольку исходные значения семафоров были равны 1 и поскольку к семафорам не было обращений со стороны других процессов, процесс A никогда не приостановится, а значения семафоров будут изменяться только между 1 и 0.

При выполнении программы с параметром 'b' процесс (B) уменьшает значения семафоров 0 и 1 в порядке, обратном ходу выполнения процесса A. Когда процессы A и B выполняются параллельно, может сложиться ситуация, в которой процесс A захватил семафор 0 и хочет захватить семафор 1, а процесс B захватил семафор 1 и хочет захватить семафор 0. Оба процесса перейдут в состояние приостанова, не имея возможности продолжить свое выполнение. Возникает взаимная блокировка, из которой процессы могут выйти только по получении сигнала.

Чтобы предотвратить возникновение подобных проблем, процессы могут выполнять одновременно несколько операций над семафорами. В последнем примере желаемый эффект достигается благодаря использованию следующих операторов:

```

struct sembuf psembuf[2];
psembuf[0].sem_num = 0;
psembuf[1].sem_num = 1;
psembuf[0].sem_op = -1;
psembuf[1].sem_op = -1;
semop(semid, psembuf, 2);

```

Psembuf - это список операций, выполняющих одновременное уменьшение значений семафоров 0 и 1. Если какая-то операция не может выполняться, процесс приостанавливается. Так, например, если значение

семафора 0 равно 1, а значение семафора 1 равно 0, ядро оставит оба значения неизменными до тех пор, пока не сможет уменьшить и то, и другое.

Если процесс выполняет операцию над семафором, захватывая при этом некоторые ресурсы, и завершает свою работу без приведения семафора в исходное состояние, могут возникнуть опасные ситуации. Причинами возникновения таких ситуаций могут быть как ошибки программирования, так и сигналы, приводящие к внезапному завершению выполнения процесса. Если после того, как процесс уменьшит значения семафоров, он получит сигнал kill, восстановить прежние значения процессу уже не удастся, поскольку сигналы данного типа не анализируются процессом. Следовательно, другие процессы, пытаясь обратиться к семафорам, обнаружат, что последние заблокированы, хотя сам заблокировавший их процесс уже прекратил свое существование. Для предотвращения подобных ситуаций предназначен флаг SEM\_UNDO. Если процесс установил этот флаг, то при завершении его работы ядро даст обратный ход всем операциям над семафорами, выполненным процессом.

Идентификатор семафора	semid		Идентификатор семафора	semid	semid
Номер семафора	0		Номер семафора	0	1
Установленное значение	1		Установленное значение	1	1

(а) После первой операции (б) После второй операции

Идентификатор семафора	semid			
Номер семафора	0		Пусто	
Установленное значение	1			

(в) После третьей операции (г) После четвертой операции

Рисунок - 4. Последовательность состояний списка структур восстановления

Ядро создает структуру восстановления всякий раз, когда процесс уменьшает значение семафора, а удаляет ее, когда процесс увеличивает значение семафора, поскольку установочное значение структуры равно 0. На Рис.4 показана последовательность состояний списка структур при выполнении программы с параметром 'а'. После первой операции процесс имеет одну структуру, состоящую из идентификатора *semid*, номера семафора, равного 0, и установочного значения, равного 1, а после второй операции появляется вторая структура с номером семафора, равным 1, и установочным значением, равным 1. Если процесс неожиданно завершается, ядро проходит по всем структурам и прибавляет к каждому семафору по единице, восстанавливая их значения в 0. В частном случае ядро уменьшает установочное значение для семафора 1 на третьей операции, в соответствии с увеличением значения самого семафора, и удаляет всю структуру целиком, поскольку установочное значение становится нулевым. После четвертой операции у процесса больше нет структур восстановления, поскольку все установочные значения стали нулевыми.

Векторные операции над семафорами позволяют избежать взаимных блокировок, как было показано выше, однако они представляют известную трудность для понимания и реализации, и в большинстве приложений полный набор их возможностей не является обязательным. Программы, испытывающие потребность в использовании набора семафоров, сталкиваются с возникновением взаимных блокировок на пользовательском уровне, и ядру уже нет необходимости поддерживать такие сложные формы системных функций.



### Задания на лабораторную работу

1. Процесс 1 порождает потомков 2 и 3, все они присоединяют к себе разделяемую память объемом ( $2 * \text{sizeof}(\text{int})$ ). Процессы 1 и 2 по очереди пишут в эту память число, равное своему номеру (1 или 2). После этого один из процессов удаляет разделяемую память, затем процесс 3 считывает содержимое области разделяемой памяти и записывает в файл. Используя семафоры, обеспечить следующее содержимое файла:

а) 1 2 1 2 1 2 1 2

б) 1 1 2 2 1 1 2 2

д) 1 2 2 1 2 2 1 2 2

2. Процесс 1 порождает потомков 2 и 3. Все процессы записывают в общую разделяемую память число, равное своему номеру. Используя семафоры, обеспечить следующее содержимое области памяти:

а) 1 2 3 1 2 3 1 2 3

б) 1 1 2 2 3 3 1 1 2 2 3 3

д) 3 1 2 3 1 2 3 1 2

Последний процесс считывает содержимое разделяемой памяти, выводит его на экран и удаляет разделяемую память.

3. Процесс 1 порождает потомков 2 и 3, все они присоединяют к себе две области разделяемой памяти M1 и M2 объемом ( $N1 * \text{sizeof}(\text{int})$ ) и ( $N2 * \text{sizeof}(\text{int})$ ) соответственно. Процесс 1 пишет в M1 число, которое после каждой записи увеличивается на 1; процесс 2 переписывает k2 чисел из M1 в M2, а процесс 3 переписывает k3 чисел из M2 в файл. После каждого этапа работы процесс 1 засыпает на t1 секунд, процесс 2 - на t2 секунд, а процесс 3 - на t3 секунд. Процессу 1 запрещается записывать в занятую область M1; процесс 2 может переписать данные, если была произведена запись в M1 и M2 свободна; процесс 3 может переписывать данные из M2, только если была осуществлена запись в M2. Используя семафоры, обеспечить синхронизацию работы процессов в соответствии с заданными условиями. Параметры N1, N2, k1, k2, k3, t1, t2, t3 задаются в виде аргументов командной строки.

4. Процесс 1 порождает потомка 2, они присоединяют к себе разделяемую память объемом ( $N * \text{sizeof}(\text{int})$ ). Процесс 1 пишет в нее k1 чисел сразу, процесс 2 переписывает k2 чисел из памяти в файл. Процесс 1 может производить запись, только если в памяти достаточно места, а процесс 2 переписывать, только если имеется не меньше, чем k2 чисел. После каждой записи (чтения) процессы засыпают на t секунд. После каждой записи процесс 1 увеличивает значение записываемых чисел на 1. Используя семафоры, обеспечить синхронизацию работы процессов в соответствии с заданными условиями. Параметры N, k1, k2, t задаются в виде аргументов командной строки.

5. Процесс 1 порождает потомков 2 и 3, все они присоединяют к себе разделяемую память объемом ( $2 * \text{sizeof}(\text{int})$ ). Процессы 1 и 2 по очереди пишут в эту память число, равное своему номеру (1 или 2). После этого один из процессов удаляет разделяемую память, затем процесс 3 считывает содержимое области разделяемой памяти и записывает в файл. Используя семафоры, обеспечить следующее содержимое файла:

а) 1 1 2 1 1 2 1 1 2

б) 2 1 1 2 1 1 2 1 1

6. Процесс 1 порождает потомков 2 и 3. Все процессы записывают в общую разделяемую память число, равное своему номеру. Используя семафоры, обеспечить следующее содержимое области памяти:

а) 1 2 1 3 1 2 1 3 1 2 1 3

б) 2 1 1 3 2 1 1 3 2 1 1 3

7. Два дочерних процесса выполняют некоторые циклы работ, передавая после окончания очередного цикла через один и тот же сегмент разделяемой памяти родительскому процессу очередные четыре строки некоторого стихотворения, при этом первый процесс передает нечетные четырехстишья, второй - четные. Циклы работ процессов не сбалансированы по времени. Родительский процесс компонует из передаваемых фрагментов законченное стихотворение и выводит его по завершении работы обоих процессов. Решить задачу с использованием аппарата семафоров.

8. Четыре дочерних процесса выполняют некоторые циклы работ, передавая после окончания очередного цикла через один и тот же сегмент разделяемой памяти родительскому процессу очередную строку некоторого стихотворения, при этом первый процесс передает 1-ю, 5-ю, 9-ю и т.д. строки, второй - 2-ю, 6-ю, 10-ю и т.д. строки, третий - 3-ю, 7-ю, 11-ю и т.д. строки, четвертый - 4-ю, 8-ю, 12-ю и т.д. строки. Циклы работ процессов не сбалансированы по времени. Родительский процесс компонует из передаваемых фрагментов законченное стихотворение и выводит его по завершении работы всех процессов. Решить задачу с использованием аппарата семафоров.

## Контрольные вопросы

15. В чем разница между двоичным и общим семафорами?
16. Чем отличаются P() и V()-операции от обычных операций увеличения и уменьшения на единицу?
17. Для чего служит набор программных средств IPC?
18. Для чего введены массовые операции над семафорами в ОС UNIX?
19. Каково назначение механизма очередей сообщений?
20. Какие операции над семафорами существуют в ОС UNIX?
21. Каково назначение системного вызова msgget()?
22. Какие условия должны быть выполнены для успешной постановки сообщения в очередь?
23. Как получить информацию о владельце и правах доступа очереди сообщений?
24. Каково назначение системного вызова shmget()?

## Литература

1. Дансмур М., Дейвис Г. Операционная система UNIX и программирование на языке Си: Пер. с англ.- М.: "Радио и связь", 1989.-192 с.
2. Забродин Л.Д. UNIX. Введение в командный интерфейс. -М.: "ДИАЛОГ-МИФИ", 1994.-144 с.
3. Керниган Б.В., Пайк Р. UNIX - универсальная среда программирования: Пер. с англ.- М.: Финансы и статистика, 1992.-304 с.
4. Робачевский А.М. Операционная система UNIX. - СПб.: BHV - Санкт-Петербург, 1997. - 528 с.
5. Т. Чан Системное программирование на C++ для UNIX. /Пер. с англ.-К.: Издательская группа BHV, 1997. - 592 с.

**ПРИЛОЖЕНИЕ Б****ПРИМЕРНЫЙ СПИСОК ЭКЗАМЕНАЦИОННЫХ ВОПРОСОВ**

1. Схема работы транслятора.
2. Описание входного языка транслятора. БНФ.
3. Формальные языки и грамматики.
4. Порождающая грамматика.
5. Классификация Хомского.
6. Операционная система UNIX.
7. Интерпретатор Shell. Команды.
8. Файловая система ОС UNIX.
9. Обыкновенные файлы. Справочники. Специальные файлы. Символические каналы связи. Расположение системы.
10. Архитектура ОС UNIX.
11. Подсистема управления файлами,
12. Процессы. Модель процесса.
13. Создание процесса. Завершение процесса.
14. Состояния процессов. Реализация процессов.
15. Потоки. Модель потоков.
16. Использование потоков.
17. Реализация потоков в пространстве пользователя.
18. Реализация потоков в ядре.
19. Смешанная реализация.
20. Активизация планировщика.
21. Всплывающие потоки.
22. Прimitives межпроцессорного взаимодействия. Семафоры и мьютексы.
23. Прimitives межпроцессорного взаимодействия. Мониторы и барьеры.
24. Сокеты Беркли
25. Распределенные системы