МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ

«БРАТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Кафедра математики и физики

УТВЕРЖДАЮ:
Проректор по учебной работе
_____Е.И.Луковникова
«_____» декабря 2018 г.

РАБОЧАЯ ПРОГРАММА ДИСЦИПЛИНЫ

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Б1.В.07

НАПРАВЛЕНИЕ ПОДГОТОВКИ

01.03.02 Прикладная математика и информатика

ПРОФИЛЬ ПОДГОТОВКИ
Инженерия программного обеспечения

Программа академического бакалавриата

Квалификация (степень) выпускника: бакалавр

СОДЕРЖАНИЕ ПРОГРАММЫ **Стр.**

1. ПЕРЕЧЕНЬ ПЛАНИРУЕМЫХ РЕЗУЛЬТАТОВ ОБУЧЕНИЯ ПО ДИСЦИПЛИНЕ, СООТНЕСЕННЫХ С ПЛАНИРУЕМЫМИ РЕЗУЛЬТАТАМИ ОСВОЕНИЯ ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ

Вид деятельности выпускника

Дисциплина охватывает круг вопросов, относящихся к научно-исследовательскому виду профессиональной деятельности выпускника в соответствии с компетенциями и видами деятельности, указанными в учебном плане.

Цель дисциплины

Целью изучения дисциплины является: ознакомление обучающихся с различными методами, приемами разработки системных программ, приемами интеграции одних программных пакетов в другие и использованию результатов интеграции при создании собственных сложных универсальных программных комплексов.

Задачи дисциплины

- изучение механизмов и алгоритмов функционирования операционных систем и их компонентов;
- освоение приемов использования системных ресурсов при разработке системных и прикладных программ;
- приобретение навыков написания программ, отвечающим требованиям безопасности.

Код компетенции	Содержание компетенций	Перечень планируемых результатов обучения по дисциплине
1	2	3
ОПК-3	Способность к разработке алгоритмических и программных решений в области системного и прикладного программирования, математических, информационных и имитационных моделей, созданию информационных ресурсов глобальных сетей, образовательного контента, прикладных баз данных, тестов и средств тестирования систем и средств на соответствие стандартам и исходным требованиям	**знать:** - основные алгоритмы решения задач **уметь:** – разрабатывать алгоритмы решения задач из области системного и прикладного программирования, математических, информационных и имитационных моделей ; **владеть:** - приемами построения алгоритмических и программных решений.
ПК-1	Способность собирать, обрабатывать и интерпретировать данные современных научных исследований, необходимые для формирования выводов по соответствующим научным исследованиям	**знать:** – способы сбора и обработки информации; **уметь:** –применять аппарат математической статистики для обработки данных; **владеть:** – методами и приемами обработки данных и интерпретации результатов

ПК-7	Способность к разработке и применению алгоритмических и программных решений в области системного и прикладного программного обеспечения	**знать:** - принципы построения программных решений; **уметь:** - разрабатывать системное и прикладного программное обеспечение **владеть:** – навыками разработки программных решений .
ПК-9	Способность составлять и контролировать план выполняемой работы, планировать необходимые для выполнения работы ресурсы, оценивать результаты собственной работы	**знать:** - принципы планирования работы по осуществлению разработки системных программ; **уметь:** – разрабатывать, оценивать и реализовывать процессы жизненного цикла программного обеспечения **владеть:** - навыками планирования, контроля и оценки результатов собственной деятельности.

2. МЕСТО ДИСЦИПЛИНЫ В СТРУКТУРЕ ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ

Дисциплина Б1.В.07 Системное программирование относится к вариативным дисциплинам и является обязательной для изучения.

Дисциплина Системное программирование базируется на знаниях, полученных при изучении таких учебных дисциплин, как: Языки и методы программирования, Теория алгоритмов.

Основываясь на изучении перечисленных дисциплин, Системное программирование представляет основу для преддипломной практики и подготовки к государственной итоговой аттестации.

Такое системное междисциплинарное изучение направлено на достижение требуемого ФГОС уровня подготовки по квалификации бакалавр.

3. РАСПРЕДЕЛЕНИЕ ОБЪЕМА ДИСЦИПЛИНЫ

3.1. Распределение объема дисциплины по формам обучения

Форма обучения	*Курс*	*Семестр*	*Трудоемкость дисциплины в часах*						*Курсовая работа*	*Вид промежуточной аттестации*
			Всего часов (с экз.)	*Аудиторных часов*	*Лекции*	*Лабораторные работы*	*Семинары Практические занятия*	*Самостоятельная работа*		
1	2	3	4	5	6	7	8	9	10	11
Очная	4	7	144	68	17	51	-	40	КР	Экзамен
Заочная	-	-	-	-	-	-	-	-	-	-

Заочная (ускоренное обучение)	-	-	-	-	-	-	-	-	-	-
Очно-заочная	-	-	-	-	-	-	-	-	-	-

3.2. Распределение объема дисциплины по видам учебных занятий и трудоемкости

Вид учебных занятий	*Трудо-емкость (час.)*	*в т.ч. в интерактивной, актив-ной, иннова-ционной формах, (час.)*	*Распределение по семестрам, час*
			7
1	2	3	4
I. Контактная работа обучающихся с преподавателем (всего)	68	54	68
Лекции (Лк)	17	6	17
Лабораторные работы (ЛР)	51	48	51
Курсовая работа*	+	-	+
Групповые (индивидуальные) консультации*	+	-	+
II. Самостоятельная работа обучающихся (СР)	40	-	40
Подготовка к лабораторным работам	8	-	8
Подготовка к экзамену в течение семестра	8	-	8
Выполнение курсовой работы	24	-	24
III. Промежуточная аттестация экзамен	36	-	36
Общая трудоемкость дисциплины час.	144	-	144
зач. ед.	4	-	4,0

4. СОДЕРЖАНИЕ ДИСЦИПЛИНЫ

4.1. Распределение разделов дисциплины по видам учебных занятий

- для очной формы обучения:

№ раз-дела и темы	*Наименование раздела и тема дисциплины*	*Трудоем-кость, (час.)*	*Виды учебных занятий, включая самостоятельную работу обучающихся и трудоемкость; (час.)*		
			учебные занятия		*самостоятельная работа обучаю-щихся*
			лекции	*лабораторные работы*	
1	2	3	4	5	6
1.	**Общие сведения об операционных системах и их структурных элементах**	**58**	**10**	**28**	**20**
1.1.	Архитектура операционных систем	12	2	6	4
1.2.	Файлы и каталоги	12	2	6	4
1.3.	Понятие процесса в системе	12	2	6	4
1.4.	Реализация многозадачности.	10	2	4	4
1.5.	Сигналы	12	2	6	4
2.	**Межпроцессное взаимодействие**	**50**	**7**	**23**	**20**
2.1	Каналы	12	2	6	4
2.2	Проблемы межпроцессного взаимодействия	7	1	2	4
2.3.	Синхронизация	11	2	5	4
2.4.	Управление памятью	9	1	4	4
2.5.	Разделяемая память	11	1	6	4
	ИТОГО	**108**	**17**	**51**	**40**

4.2. Содержание дисциплины, структурированное по разделам и темам

№ раздела и темы	*Наименование раздела и темы дисциплины*	*Содержание лекционных занятий*	*Вид занятия в интерактивной, активной, инновационной формах, (час.)*
1	**2**	**3**	**4**
1.	**Общие сведения об операционных системах и их структурных элементах**		
1.1.	Архитектура операционных систем	Ядро операционной системы. Типы архитектур ядер операционных систем. Ядро Linux и его особенности . Специфические особенности программирования работы ядра в	-

		сравнении с пользовательской программой.	
1.2.	Файлы и каталоги	Абстракция файла. Режим файла и режим доступа. Файловый ввод-вывод. Каталоги. Дерево каталогов. Операции с каталогами. Узлы устройств.	Проблемная лекция(1 час)
1.3.	Понятие процесса в системе	Программы, процессы и потоки. Атрибуты процесса. Типы процессов. Состояние процесса. Порождение нового процесса fork(). Передача управления процессу: семейство exec*().	-
1.4.	Реализация многозадачности.	Основные цели планирования. Планировщик. Системы с кооперативной многозадачностью и системы с преемптивной многозадачностью. Основные алгоритмы планирования.	Проблемная лекция(1 час)
1.5.	Сигналы	Виды сигналов. Порождение сигнала. Перехват сигнала. Обработчики сигнала.	Проблемная лекция(1 час)
2.	**Межпроцессное взаимодействие**		
2.1	Каналы	Неименованные каналы. Системный вызов pipe(). Чтение и запись в неименованный канал. Файлы FIFO. Создание файла. Работа с FIFO.	
2.2	Проблемы межпроцессного взаимодействия	Конкуренция процессов за ресурсы. Критические области. Тупиковые состояния. Модели взаимоблокировок.	Проблемная лекция(1 час)
2.3.	Синхронизация	Средства синхронизации в ядре. Семафоры и мьютексы. Атомарные операции. Средства синхронизации в пространстве пользователя.	Проблемная лекция(1 час)
2.4	Управление памятью	Адресное пространство процесса. Управление динамической памятью приложения. Организация памяти в системе. Интерфейсы для работы с памятью в ядре.	Проблемная лекция(1 час)
2.5	Разделяемая память	Отображение файла на память. Анонимные отображения в памяти. Использование разделяемой памяти. Блокировка участка памяти.	

4.3. Лабораторные работы

№ п/п	*Номер раздела дисциплины*	*Наименование лабораторной работы*	*Объем (час.)*	*Вид занятия в интерактивной, активной, инновационной формах, (час.)*
1	**1.**	Аргументы программы	6	Тренинги в малой группе (6 час)
2		Низкоуровневые операции ввода-вывода	6	Проектная деятельность (6 час)
3		Файловый ввод/вывод	6	Проектная деятельность (6 час)
		Процессы: порождение и управление	6	Проектная деятельность (6 час)
4		Сигналы	6	Проектная деятельность (6 час)
6	**2.**	Неименованные каналы	6	Проектная деятельность (6 час)
7		Файлы FIFO	3	-
8		Отображение файлов на память.	6	Проектная деятельность (6 час)
9		Разделяемая память.	6	Проектная деятельность (6 час)
		ИТОГО	**51**	**48**

4.4. Семинары/ практические занятия

учебным планом не предусмотрено.

4.5. Контрольные мероприятия: курсовая работа

Курсовая работа выполняется как индивидуальное домашнее задание. Зачтенные работы оформляются и включаются в портфолио обучающегося.

Цель: Углубление знаний в области системного программирования, формирование навыков самостоятельной работы по дисциплине.

Структура:

1. Титульный лист.
2. Содержание.
3. Введение.
4. Глава 1 (теоретическая).
5. Глава 2 (описание разработки).
6. Заключение.
7. Литература.
8. Приложения (при необходимости)

Основная тематика: Использование и управление системными ресурсами Unix-подобных операционных систем.

Темы курсовых работ:

1. Управление потоками в семействе *Nix
2. Алгоритмы планирования задач
3. Работа с областями памяти
4. Управление памятью в ядре Linux
5. Страничный кэш и обратная запись страниц
6. Моделирование проблемных ситуаций межпроцессного взаимодействия

7. Механизмы использования барьеров в межпроцессном взаимодействии
8. Сигналы реального времени в семействе *Nix
9. Использование разделяемой памяти для организации межпроцессного взаимодействия
10. Использование общих файлов для организации межпроцессного взаимодействия
11. Обработка ошибок и перехват исключений
12. Перехват и обработка прерываний семействе *Nix
13. Защита разделяемых ресурсов спин-блокировками
14. Программные средства защиты от несанкционированного доступа
15. Программирование виртуальной консоли
16. Работа с таблицей процессов
17. Разработка утилит групповой работы с файлами
18. Использование каналов для передачи данных между процессами
19. Виртуальные файловые системы
20. Виртуальная память
21. Программа поиска файлов-дублей
22. Разработка командной оболочки
23. Особенности программирования процессов-демонов
24. Алгоритмы работы с двоичными деревьями
25. Написание защищенных программ
26. Чтение и анализ системных журналов
27. Событийно-орентированное программирование

Рекомендуемый объем: 25-30 стр.

Выдача задания, защита курсовых работ проводится в соответствии с календарным учебным графиком.

Оценка	**Критерии оценки курсовой работы**
отлично	Задача на разработку решена полностью. Программа работает устойчиво, устойчиво, в ней предусмотрена обработка ошибок и исключений и - работа содержит грамотно изложенную теоретическую базу с последовательным и аргументированным изложением, обоснованными выводами и предложениями по использованию полученных результатов и - работа оформлена в соответствии с нормативными требованиями, предъявленными к подобным материалам и -работа не содержит грамматических ошибок, опечаток, неаккуратных исправлений и - уникальность текстовой части работы не ниже 60% и - при защите обучающийся четко, ясно, последовательно излагает суть работы, уверенно отвечает на вопросы.
хорошо	Задача на разработку решена полностью, однако программа не обрабатывает возникающие исключения и содержит грамотно изложенную теоретическую базу с последовательным и аргументированным изложением, обоснованными выводами и предложениями по использованию полученных результатов; и/или -при оформлении работы допущены опечатки, некоторые элементы оформления не соответствуют предъявленным требованиям; -уникальность текстовой части работы не ниже 60%; - при защите обучающийся показывает знание темы, последовательно излагает суть работы, без особых затруднений отвечает на вопросы.
удовлетворительно	Задача на разработку решена не полностью. Реализованы не менее половины требуемых функций.

	и/или - работа содержит теоретическую базу, но отличается поверхностным анализом проблем или просто их перечислением без соответствующего анализа, в ней просматриваются непоследовательность изложения и отсутствие описания или анализа собственных результатов, в работе содержатся невнятные выводы и предложения; и/или - при оформлении работы допущены опечатки, некоторые элементы оформления не соответствуют предъявленным требованиям; - уникальность текстовой части работы от 40% до 60% - при защите обучающийся показывает поверхностные знания, на вопросы отвечает неуверенно.
неудовлетворительно	Задание на разработку решено менее, чем на 50% или уникальность текстовой части работы менее 40% или при защите обучающийся обнаруживает незнание большей части темы или совсем не ориентируется в ней, отвечает на вопросы бессистемно, неуверенно, неправильно или же обучающийся вовсе не явился на защиту без уважительной причины.

5. МАТРИЦА СООТНЕСЕНИЯ РАЗДЕЛОВ УЧЕБНОЙ ДИСЦИПЛИНЫ К ФОРМИРУЕМЫМ В НИХ КОМПЕТЕНЦИЯМ И ОЦЕНКЕ РЕЗУЛЬТАТОВ ОСВОЕНИЯ ДИСЦИПЛИНЫ

Компетенции / ***№, наименование разделов дисциплины***	***Кол-во часов***	***Компетенции***				***Σ комп.***	$t_{ср}$, ***час***	***Вид учебных занятий***	***Оценка результатов***
		ОПК	***ПК***						
		3	***1***	***7***	***9***				
1	**2**	**3**	**4**		**5**	**6**	**7**	**8**	**9**
1. Общие сведения об операционных системах и их структурных элементах	58	+	+	-	-	2	29	Лк, ЛР	экзамен
2. Межпроцессное взаимодействие	50	-	-	+	+	2	25	Лк, ЛР	КР, экзамен
всего часов	**108**	**29**	**29**	**25**	**25**	**4**	**27**		

6. ПЕРЕЧЕНЬ УЧЕБНО-МЕТОДИЧЕСКОГО ОБЕСПЕЧЕНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ ОБУЧАЮЩИХСЯ ПО ДИСЦИПЛИНЕ

1. Ратинская, Е. В. Системное программирование : методические указания и задания к выполнению лабораторных работ / Е. В. Ратинская. - Братск : БрГУ, 2015. - 99 с.

7. ПЕРЕЧЕНЬ ОСНОВНОЙ И ДОПОЛНИТЕЛЬНОЙ ЛИТЕРАТУРЫ, НЕОБХОДИМОЙ ДЛЯ ОСВОЕНИЯ ДИСЦИПЛИНЫ

№	*Наименование издания*	*Вид занятия*	*Количество экземпляров в библиотеке*, шт.	*Обеспеченность*, (*экз./ чел.*)
1	2	3	4	5
	Основная литература			
1.	Лав, Р. Linux. Системное программирование: учебное пособие / Р. Лав; Пер. с англ. - 2-е изд. - СПб. : Питер, 2014. – 448 с.	Лк, ЛР, СР	10	0,5
2	Флоренсов, А.Н. Системное программное обеспечение : учебное пособие / А.Н. Флоренсов ; Минобрнауки России, Омский государственный технический университет. - Омск : Издательство ОмГТУ, 2017. - 139 с. - Библиогр. в кн. - ISBN 978-5-8149-2441-4 ; То же [Электронный ресурс]. - URL:http://biblioclub.ru/index.php?page=book&id=493301	Лк,СР	ЭР	1
	Дополнительная литература			
3	Гриценко, Ю.Б. Системы реального времени : учебное пособие / Ю.Б. Гриценко ; Томский Государственный Университет Систем Управления и Радиоэлектроники (ТУСУР), Кафедра автоматизации обработки информации (АОИ). - Томск : ТУСУР, 2017. - 253 с. : ил. - Библиогр. в кн. ; То же [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=481015	СР	ЭР	1
4	Молчанов, А. Ю. Системное программное обеспечение : учебник для вузов / А. Ю. Молчанов. - Санкт-Петербург : Питер, 2006 - 396 с. - (Учебник для вузов).	ЛР, КР, СР	36 включая аналоги	1
5	Царев, Р.Ю. Программирование на языке Си : учебное пособие / Р.Ю. Царев ; Министерство образования и науки Российской Федерации, Сибирский Федеральный университет. - Красноярск : Сибирский федеральный университет, 2014. - 108 с. : табл., схем. - Библиогр. в кн. - ISBN 978-5-7638-3006-4 ; То же [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=364601	СР	ЭР	1

8. ПЕРЕЧЕНЬ РЕСУРСОВ ИНФОРМАЦИОННО-ТЕЛЕКОММУНИКАЦИОННОЙ СЕТИ «ИНТЕРНЕТ» НЕОБХОДИМЫХ ДЛЯ ОСВОЕНИЯ ДИСЦИПЛИНЫ

1.Электронный каталог библиотеки БрГУ
http://irbis.brstu.ru/CGI/irbis64r_15/cgiirbis_64.exe?LNG=&C21COM=F&I21DBN=BOOK&P21DBN=BOOK&S21CNR=&Z21ID=.
2. Электронная библиотека БрГУ
http://ecat.brstu.ru/catalog .
3. Электронно-библиотечная система «Университетская библиотека online»
http://biblioclub.ru .
4. Электронно-библиотечная система «Издательство «Лань»
http://e.lanbook.com .
5. Информационная система "Единое окно доступа к образовательным ресурсам"
http://window.edu.ru .
6. Научная электронная библиотека eLIBRARY.RU http://elibrary.ru .
7. Университетская информационная система РОССИЯ (УИС РОССИЯ)
https://uisrussia.msu.ru/ .
8. Национальная электронная библиотека НЭБ
http://xn--90ax2c.xn--p1ai/how-to-search /.
Специальные тематические сайты
1. Сайт по программмированию http://life-prog.ru ;
2. Электронный журнал “Типичный програмст” https://tproger.ru .
3. Сайт по программмированию https://professorweb.ru
4. Сайт по программмированию https://metanit.com

9. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОБУЧАЮЩИХСЯ ПО ОСВОЕНИЮ ДИСЦИПЛИНЫ

Обучающийся должен разработать собственный режим равномерного освоения дисциплины. Подготовка студента к предстоящей лекции включает в себя ряд важных познавательно-практических этапов:

– чтение записей, сделанных в процессе слушания и конспектирования предыдущей лекции, вынесение на поля всего, что требуется при дальнейшей работе с конспектом и учебником;

– техническое оформление записей (подчеркивание, выделение главного, выводов, доказательств);

– выполнение практических заданий преподавателя;

– знакомство с материалом предстоящей лекции по учебнику и дополнительной литературе.

Успешность выполнения лабораторных работ определяется подготовкой к ним. Подготовка к лабораторным работам содержит:

- изучение теоретического материала, содержащегося в учебной литературе, изучение лекционного материала,

- знакомство с заданиями на лабораторную работу;

- составление плана выполнения лабораторной работы.

Наиболее продуктивной является самостоятельная работа в библиотеке, где доступны основные и дополнительные печатные и электронные источники.

При выполнении приведенных выше рекомендаций подготовка к экзамену сведется к повторению изученного и совершенствованию навыков применения теоретических положений и различных методов решения к стандартным и нестандартным заданиям.

9.1. Методические указания для обучающихся по выполнению лабораторных работ

Лабораторная работа №1

Аргументы программы

Цель работы:

Изучение синтаксиса аргументов главной функции программы, приёмов их использования.

Теоретические сведения:

1. Функция main и её аргументы

При вызове некоторых программ из командной строки после имени программы можно указывать ключевые слова, влияющие на результат работы.

К примеру, хорошо известная вам программа компиляции `gcc` может вызываться с разными аргументами:

```
gcc –o результат исходник.c
gcc исходник.c –lm.
```

Слова, идущие после имени программы, называются *аргументами программы*.

Аргументы могут оказаться очень полезными и при вызове одной программы из другой. Таким способом программы могут обмениваться данными, необходимыми им в работе. Так, вызванная программа может получить от вызывающей информацию о каталоге, с которым её необходимо работать, правах доступа к файлам, идентификаторах системных ресурсов и т.д., словом, всю ту информацию, которую она не может запросить напрямую у пользователя.

Множество аргументов программы можно разделить на два класса:

- *Опции*. Их обычно используют для управления режимами работы программы. Опции, в свою очередь, подразделяют на длинные (многосимвольные) и короткие (односимвольные). Короткие опции выделяются дефисом, длинные – двумя дефисами перед названием. Существуют соглашения по использованию опций. Например, опции `–h` и `--help` вызывают справку по программе.

- *Параметры*. Используют для передачи дополнительной информации в программу. Параметры могут быть названы как угодно. Параметры подразделяют на *зависимые* и *свободные*. Зависимые параметры привязаны каждый к своей опции, после которой и указываются. Свободные не связаны напрямую с опциями.

Рассмотрим, например, вызов компилятора gcc:

```
gcc –o результат исходник.c
```

Здесь

`gcc` – имя программы;

`-o` – короткая опция;

`результат` – зависимый аргумент опции -o;

`исходник.c` – свободный аргумент программы.

2. Передача аргументов в программу

Программа не запрашивает свои аргументы у пользователя обычным порядком[1]. Она получает их уже на старте. Аргументы сбрасываются в функцию main() в виде текстовой строки.

Для этого заголовок функции main должен иметь вид:

```
int main (int argc, char *argv[],char **env)
```

или

```
int main (int argc, char**argv, char **env).
```

Здесь

- **argc** сохраняет общее число аргументов программы.

[1] т.е.через функцию ввода, такую, как scanf

- **argv** представляет собой указатель на символьный массив слов - аргументов. При этом следует помнить что argv[0] формально содержит имя программы. Оканчивается массив argv пустым указателем NULL.
- **env** – указатель на массив строк, описывающих текущее окружение процесса. Этот аргумент можно пропустить.

Вместо имен argv, argc и env допускается использование любых других. В качестве примера рассмотрим программу echo1, которая выводит на экран переданную ей строку аргументов.

Пример 1.1. Вывод на печать строки аргументов программы echo1:

```
1  #include <stdio.h>
2  main (int argc, char *argv[])  {
3     int i;
4     for (i=1; i<argc; i++)
5        printf("%s ", argv[i]);
6  }
```

Теперь, если вызвать программу командой

```
echo1  one two three ,
```

то в результате своей работы она напечатает

```
one two three
```

Слова "one", "two", "three" – это и есть аргументы нашей программы.

При запуске программы система запишет их в массив argv:

argv={"echo1", "one", "two", "three", NULL},

В переменную argc будет передано количество аргументов (argc=4).

Чтобы распечатать аргументы, нужно извлечь их по одному из массива. Здесь argv[1]="one", argv[2]="two", argv[3]="three".

Итак, что нам нужно помнить об аргументах программы?

Во-первых, переменная argc и массивы argv и env не требуется объявлять в программе. Они даны вам готовенькими.

Во-вторых argv и env – это всегда строковые массивы, которые состоят из слов.

Даже если вы вызываете вашу программу с числовыми аргументами, например,

```
progr  12 34 56,
```

для неё 12, 34, и 56 – все равно не числа, а слова и, следовательно, над ними нельзя производить никаких математических действий.

Если вы хотите, чтобы над аргументами программы можно было выполнять математические операции, их необходимо преобразовывать в числовой формат.

Для этого в заголовочном файле **stdlib.h** объявлены функции **atoi(), atof()** и **atol()**:

1) int atoi(const char *s) преобразовывает строки в число типа int (целое). При неудачном преобразовании вернет 0.

2) double atof(const char *s) преобразовывает строки, в представляемое ей число типа double.

3) long atol(const char *s) преобразовывает строки в число типа long (длинное целое).

В качестве примера,напишем программу, которая находит корни квадратного уравнения, по коэффициентам a, b, c, заданным в виде строки аргументов.

Пример 1.2. Преобразование аргументов и работа с ними

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  int main (int argc, char *argv[])
5  {
6     if (argc!=4)
7     {
```

```
8       printf("Неверное количество аргументов \n");
9       return 1;
10    }
11    float a, b,c;
12    a=atof(argv[1]);   // Преобразование аргумента
13    b=atof(argv[2]);   //  из строкового формата
14    c=atof(argv[3]);   //  в числовой
15    float  dscr, x1, x2;
16    dscr=a*a-4*b*c;
17    if (dscr<0)
18       printf ("Действительных корней нет  \n");
19    else if (dscr= =0)
20    {
21       x1=(-1)*b/(2*a);
22       printf("Корни одинаковы: x1=x2=%f \n", x1);
23    }
24    else
25    {
26       dscr=sqrt(dscr);
27       x1=(-1*b+dscr)/(2*a);
28       x2=(-1*b-dscr)/(2*a);
29       printf("Корни уравнения: x1=%f, x2=%f\n", x1, x2);
30    }
31    return 0;
32  }
```

Задание:

Вариант 1.

1. Напишите программу, которая принимает в качестве аргумента целое число и находит все его делители.

2. Видоизмените предыдущую программу так, чтобы, в зависимости от параметра опции, могли выводиться справка по программе и промежуточные результаты.

Вариант 2.

1. Напишите программу, которая принимает в качестве аргумента слово и букву и пределяет, есть ли эта буква в составе слова.

2. Видоизмените предыдущую программу так, чтобы, в зависимости от переданной опции, определялись, сколько раз указанная буква входит в слово, на каких позициях в слове она находится. Передаваемую букву тоже задайте как зависимый параметр опции.

Вариант 3.

1. Напишите программу для расчета значений функции $f(x) = x^2 + 3x - 1$ в указанном диапазоне изменения аргумента [a, b] с шагом h. Параметры a b, и h – аргументы программы.

2. Видоизмените предыдущую программу так, чтобы параметры a b, и h передавались через зависимые аргументы опций.

Вариант 4.

1. Напишите программу, которая в качестве аргументов принимает строку и печатает из неё каждое второе слово.

2. Напишите программу которая принимает два числовых аргумента и в зависимости от опции находит:

- сумму;
- разность;
- произведение.

Вариант 5.

1. Напишите программу, которая принимает в командной два слова сравнивает их в лексикографическом порядке.

2. Видоизмените предыдущую программу так, чтобы, в зависимости от параметра опции, могли выводиться справка и промежуточные результаты.

Вариант 6.

1. Напишите программу, которая составляет квадратное уравнение по известным корням. Корни уравнения задаются в строке аргументов программы.

2. Видоизмените предыдущую программу так, чтобы, в зависимости от параметра опции, выводилось либо само уравнение, либо уравнение и дискриминант, либо справка по программе.

Вариант 7.

1. Напишите программу, которая вычисляет суммарную длину слов, введенных в командной строке.

2. Видоизмените предыдущую программу так чтобы, в зависимости от параметра опции, могла выводиться длина каждого слова.

Вариант 8.

1. Напишите программу, которая принимает в командной строке до 5 слов и выводит на экран все слова, начинающиеся с буквы, указанной пользователем.

2. Видоизмените предыдущую программу так, чтобы, в зависимости от переданной опции печатались либо все слова, либо только одно. Передаваемую букву тоже задайте как зависимый параметр опции.

Вариант 9.

1. Напишите программу, которая принимает в командной строке слово *a* и печатает это слово несколько раз подряд, каждый раз укорачивая слово на одну букву.

2. Видоизмените предыдущую программу так, чтобы, в зависимости от опции, слово могло укорачиваться либо с конца, либо с начала.

Вариант10.

1. Напишите программу, печатающую свои аргументы в обратном порядке.

2. Напишите программу, которая предлагает пользователю 5 примеров по арифметике и в зависимости от предоставленной ей опции может работать в одном из двух режимов:

1) В тестовом – после каждого вопроса выводится правильный ответ.

2) В простом режиме - правильные ответы не выводятся.

Порядок выполнения:

1. Изучить теоретические сведения.
2. Изучить примеры программ, приведенных в теоретической части.
3. Модифицировать пример 1.1 или 1.2 для выполнения задания 1 согласно Вашему варианту.
4. Модифицировать пример 1.3 для выполнения задания 2 согласно Вашему варианту.
5. Оформить отчет.

Форма отчетности:

Отчет по работе содержит:

1. Наименование лабораторной работы;
2. Разработанную программу;
3. Результаты её тестирования;
4. Выводы по работе.

Основная литература

1. Лав, Р. Linux. Системное программирование: учебное пособие / Р. Лав; Пер. с англ. - 2-е изд. - СПб. : Питер, 2014.

Дополнительная литература

2. Молчанов, А. Ю. Системное программное обеспечение : учебник для вузов / А. Ю. Молчанов. - Санкт-Петербург : Питер, 2006.

Контрольные вопросы для самопроверки

1. Для чего нужны аргументы программы?
2. Какую смысловую нагрузку несет аргумент argc?
3. Что содержит argv[0]?
4. Можно ли в вызове функции main указать аргументы, отличные от int argc, char**argv и char**env?
5. Для чего используются опции программы?
6. Какие значении может возвращать функция getopt()?
7. Что означает строка "aht:s:", указанная в качестве третьего аргумента функции getopt()?
8. Какое значение сохранится в переменой optind при вызове программы с аргументами -t 23 -s qwerty -a?

Лабораторная работа 2.

Низкоуровневые операции ввода-вывода

Цель работы: Изучить синтаксис низкоуровневых функций ввода и вывода. Научиться создавать простейшие программы, использующие низкоуровневый ввод- вывод.

Теоретические сведения

1. Общие сведения

Любая программа должна получать или передавать информацию, т.е. осуществлять операции ввода-вывода. Системные программы для этих целей используют низкоуровневые операции ввода-вывода. Низкоуровневые функции, в отличие от высокоуровневых, напрямую обращаются к системным вызовам. Это даёт значительную экономию времени на каждый цикл чтения-записи. Кроме того, программы ядра попросту не имеют доступа к высокоуровневым библиотекам.

Ещё одной особенностью низкоуровневых операций ввода-вывода является то, что они при обращении к файлу используют файловые дескрипторы.

Файловый дескриптор – это целое число, которое идентифицирует открытый файл в программе.

При открытии файла ему назначается первый незанятый дескриптор, по которому программа и распознаёт файл. Следует отметить, что при открытии одного файла разными программами, его дескрипторы в этих программах могут не совпадать.

2. Операции чтения и записи

Операции чтения и записи осуществляются функциями read() и write() соответственно:

ssize_t read(int fd, void *buf, size_t count);

ssize_t write(int fd, const char *buf, size_t count);,

где

fd – файловый дескриптор.

Поскольку консоль (терминал и клавиатура) тоже считается файлом, функции read() и write() могут обращаться и к ней. Для этого в системе зарезервированы стандартные номера файловых дескрипторов:

fd=0 означает ввод с клавиатуры,

fd=1– вывод на экран,

fd=2 – запись в стандартный поток ошибок.

Прочие же значения fd связывают поток с открытым файлом и назначаются автоматически при вызове open().

buf – указатель на строку, содержащую передаваемую информацию.

count – количество символов с строке *buf.

Обе функции возвращают целое число (тип ssize_t обычно целый) – количество символов, которые удалось прочитать или записать. Если при выполнении функции произошел сбой, будет возвращено значение -1.

Пример 2.1. Программа, выводящая сообщение ”Hello, word!”на экран

```
main()
{
write (1,”Hello, word!/n”,9);
}
```

Функция write(), также как и read(), не требует подключения специальной библиотеки, поскольку является встроенной функцией языка СИ. Рассмотрим её вызов. Первый аргумент равен 1, что соответствует дескриптору стандартного вывода (терминалу), далее идет константная строка, которая должна быть напечатана и последним аргументом – количество символов в этой строке. При этом символ конца строки '/n' здесь считается за два. Для корректной работы функции write(), между аргументами не должно быть пробелов.

Мы можем использовать функцию write() для вывода строк, не являющимися константными. В качестве примера, считаем с терминала слово word и напечатаем его задом наперед.

Для того чтобы работать с частями word, нам нужно задавать его как указатель в виде массива.

Массив можно задавать 2 способами:

```
char word [n];
char * word =(char*)calloc (n, sizeof (char));
```

Функция calloc () выделяет область памяти и получает указатель на её начало. Поскольку в прототипе функции write() используется указатель *buf, при задании второго аргумента мы должны писать адрес в памяти, где содержится наше слово. Это тоже можно сделать двумя способами:

Если требуется напечатать i-ю букву или фрагмент слова длиной len, начиная с i-й буквы, обращаемся по адресу этой буквы. Операция взятия адреса обозначается в Си амперсандом (&):

```
    write( 1,&word[i], 1) ;
write( 1,&word[i], len);
```

Если нужно вывести слово целиком или его фрагмент, начиная с первой (текущей) буквы, записываем вторым аргументом указатель на массив:

```
write(1, word, strlen(word));
write(1, word, len);.
```

Пример 2.2. Печать слова задом наперед

```
#include <malloc.h>
main(){
const int n=60;
char* word=(char*)calloc(n,sizeof(char));
int len=0;
write (1,”Print anything\n ”,14);
/* Запрашиваем слово длиной не более n символов. В
 /* переменную len сохраняется фактическая
 /* длина слова: */
len=read(0,word,n);
int i;
/* Печатаем слово в обратном порядке */
for (i=len-1;i>=0;i--){
  write (1, &word[i],1);
 }
}
```

Рассмотрим пример программы-«заики», печатающей некоторые буквы более одного раза.

Пример 2.3. Программа-заика

```
#include <malloc.h>
 main(){
const int n=60;
 char * word=(char*)calloc (n,sizeof(char));
write (1,"Print anything\n ",14);
read(0,word,n);
while (*(word+2))     // пока есть хотя бы  еще 3 буквы
{
   write (1, word,3);   // печатаем по три буквы
   word+=2;          // сдвигаемся на две буквы
}
}
```

В этой программе обращение к переменной word посредством указателей. Собственно, сама переменная является указателем на массив букв. Для перемещения по слову мы используем операции сдвига:

word+=2 сдвиг в памяти на 2*sizeof (тип) ячеек и
доступ по адресу, находящемуся на 2 элемента «правее» текущего;

Для получения текущего значения буквы, на которую указывает word применяется операция разадресации(*word) .

Программа «заикается» благодаря строке 8. В ней последовательно выводятся куски слова по три буквы, после чего в строке 9 осуществляется сдвиг по слову на две буквы правее.

Контрольные вопросы

Какие операции называют операциями ввода-вывода?
Какие функции называют низкоуровневыми?
Почему системные программы используют низкоуровневый ввод-вывод?
Каким значениям файлового дескриптора соответствует стандартный поток вывода?
Каким значениям файлового дескриптора соответствует стандартный поток ввода?
Какую информацию возвращает функция read()?
Как можно объявить строку для работы с функциями read() и write()?
Какие операции с указателями Вы знаете?

Порядок выполнения работы

Изучить теоретические сведения.
Изучить примеры программ, приведенных в теоретической части.
Модифицировать пример 2.2 или 2.3 для выполнения задания согласно варианту.
Оформить отчет.

Отчет должен включать:
– титульный лист;
– формулировку цели работы;
– формулировку задания;
– листинги программ;
– результаты выполнения программ;
– выводы, согласованные с целью работы;
– ответы на контрольные вопросы.

Форма отчетности:

Отчет по работе содержит:
1. Наименование лабораторной работы;
2. Разработанную программу;
3. Результаты её тестирования;
4. Выводы по работе.

Основная литература

1. Лав, Р. Linux. Системное программирование: учебное пособие / Р. Лав; Пер. с англ. - 2-е изд. - СПб. : Питер, 2014.

Дополнительная литература

2. Молчанов, А. Ю. Системное программное обеспечение : учебник для вузов / А. Ю. Молчанов. - Санкт-Петербург : Питер, 2006.

Лабораторная работа 3

Файловый ввод-вывод

Цель работы: Изучить синтаксис низкоуровневых функций, предназначенных для работы с файлами. Освоить способы работы с файлами.

Теоретические сведения

1. Модель файловой системы Linux/Unix

UNIX-подобные операционные системы строятся на двух абстракциях: файлы и процессы. Файлом называют любое хранилище информации: обычные файлы, каталоги, каналы для передачи информации, устройства, буферы памяти.

Файл – это линейный поток байтов.

Работу с файлами можно описать четырьмя словами: открыть, прочитать, записать, закрыть.

2. Права доступа

Прежде познакомимся с правами доступа к файлу.

Для каждого файла определены три группы пользователей:

- владелец файла (u, user);
- группа, к которой принадлежит владелец(g, group);
- прочие пользователи (o, other).

Права доступа к файлу записывают в виде целого числа вида 0ugo, где каждая из цифр u, g, o означает определённую группу пользователей и может принимать значение от 0 до 7 в зависимости от назначаемых прав доступа:

1- на исполнение;

2- на чтение;

4- на запись.

Комбинированные права получают, суммируя нужные значения. Например, чтобы разрешить одновременно чтение и запись, в соответствующей позиции пишут 6 (2 + 4).

Таким образом, права на чтение, запись и исполнение для владельца, на чтение и исполнение для группы и отсутствие прав доступа для всех остальных пользователей запишутся так: 0750.

3. Создание файла

Первым шагом организации работы с файлом в программе всегда является объявление переменной, в которой будет хранится дескриптор файла:

```
int fd;
```

Далее создаётся новый файл. Для этого можно использовать одну из функций creat() или open().

Функция creat() применяется в случае, если созданный файл не планируется открывать сразу же:

```
int creat(int const *filename, mode_t mode);.
```

Здесь

filename - имя файла;

mode - режим доступа к файлу.

Если в программе предполагается осуществлять операции ввода/вывода в файл, то его можно создать функцией open().

4. Открытие файла

Прежде чем начать записывать данные в файл или считывать их оттуда, его нужно открыть. Для этого в языке Си имеется функция open():

int open (const char * fiilename, int flags, mode_t mode);

где

filename – имя файла.

flags – флаг цели. Указывает, для чего открывается файл:

O_RDONLY – только для чтения,

O_WRONLY – только для записи,

O_RDWR – для чтения/записи,

O_CREAT – создать файл, если он не существует,

O_TRUNC – очистить при открытии,

O_APPEND – дозапись в конец файла,

O_EXCL – не открывать, если файл уже существует.

Флаги суммируются с помощью побитовой операции 'И' (обозначается '|').

Сами флаги представляют собой целые значения. Обозначения, указанные выше, не что иное, как подстановочные константы, объявленные в заголовочном файле fcntl.h.

mode – режим доступа в формате 0ugo.

Например, фрагмент кода

int fd;

fd=open (“my_file.dat”, O_RDWR|O_APPEND, 0763);

выполняет открытие файла с именем “my_file.dat”, для чтения и дозаписи в конец файла. При этом владелец может все: читать, писать и запускать файл на исполнение (7=1+2+4), члены группы владельца – читать и писать (6=2+4), но не могут запускать файл, а прочие пользователи могут запускать файл и записывать в него.

Согласно общепринятому правилу, функция возвращает отрицательное значение в случае ошибки. В случае успеха возвращается дескриптор открытого файла.

5. Закрытие файла

Закрытие файла и связанного с ним потока байтов осуществляется функцией close():

close (fd);

или

(int)ret=close (fd);.

Здесь ret =0, если закрытие прошло без ошибок и ret= -1, если не удалось.

Функция close() освобождает файловый дескриптор, делая его доступным для использования другими файлами.

6. Чтение из файла и запись в него

Для чтения и записи применяют функции read() и write(), описанные в предыдущей лабораторной работе.

7. Доступ к произвольному месту в файле

При открытии файла, система автоматически создаёт указатель позиции чтения-записи, устанавливая его либо в начало, либо в конец (если указан флаг O_APPEND) файла. Каждая операция чтения-записи смещает этот указатель на переданное число байт. Если нужно получить доступ к определённому месту в файле, используют функцию lseek():

off_t lseek (int fd, off_t seek, int whence);,

где

fd – дескриптор открытого файла;

seek – смещение на seek позиций. Может принимать как положительные, так и отрицательные значения;

whence – точка отсчета для смещения. Может принимать одно из трёх значений:

SEEK_SET начало файла;

SEEK_CUR текущая позиция;

SEEK_END конец файла.

Тип off_t обычно совпадает с long int, но может принимать и другие значения.

8. Примеры программ

Рассмотрим пример программы, помещающей в файл с названием “myfile.doc” строку ”Hello,word!”

Пример 3.1. Программа, которая создает файл и записывает в него строку.

```
#include <fcntl.h>   // Для флагов open()
#include <string.h>
main(){
int fd;
char message[]=”Hello, word!”;
int length=strlen(message);
fd=open (“myfile.doc”, O_WRONLY|O_CREAT,0666);
write(fd, message, length);
close(fd);
}
```

В первой же строке этой программы подключается заголовочный файл fcntl.h. Он нужен для распознавания флагов (flags) функции open(). Переменная fd, объявленная в строке 4 используется для хранения дескриптора открытого файла.

Строка 5 подготавливает текст для записи, занося его в строковую переменную message. Сам файл открывается в строке 7 командой

fd=open (“myfile.doc”, O_WRONLY|O_CREAT,0666);.

Как видно по аргументам функции open(), файл открыт для простой записи (O_WRONLY) и будет создан, если система не найдёт его в текущей папке (O_CREAT). При создании ему назначаются права доступа к чтению и записи всеми группами пользователей (0666).

В строке 8 производится запись в этот файл содержимого message. При этом функции write() в качестве целевого потока вывода мы указываем дескриптор fd.

Довольно часто возникает задача прочитать файл до конца. Основной сложностью здесь является то, что мы заранее не знаем, какое количество символов он содержит. Если мы попытаемся вызвать функцию read() в бесконечном цикле, т.е. "пока читается", то программа неизбежно даст сбой. Чтобы этого избежать, используем тот факт, что read() может возвращать целое значение – количество прочитанных символов. Если очередной символ ей прочитать не удалось, то функция вернет 0 или -1.

В следующем примере организовано чтение из файла посимвольно:

Пример 3.2. Чтение из файла “myfile.doc” .

```
#include <unistd.h>
#include <fcntl.h>
void exit(int);  /*Прототип нужен, чтобы компилятор не
                   ругался*/
main(int arc, char *arv[]){
    if(arc<2){
      write(1,"Вы забыли указать имя текстового файла",71);
       exit (1);
     }
int fd;
char letter[1];     // Читать будем по буквам
if((fd=open(arv[1], O_RDONLY, 0220))<0) exit(1);
while ((read(fd, letter,1)==1)   // Читаем, пока читается
{
   write (1, letter,1);
}
close(fd);
}
```

Имя открываемого файла здесь описано как аргумент программы arv[1].

Конструкция if в строке 12 проверяет, что запрашиваемый файл действительно существует и может быть открыт. Если файл открыть не удастся, программа завершит свою работу (exit(1)).

В строке 13 функция read() пытается считать очередной символ и возвращает значение 1, если это ей удалось и 0 – если нет. Цикл while переходит на новую итерацию, если возвращенное значение было ненулевым.

Чтение из файла посимвольно не очень хорошая идея, поскольку требует значительных расходов времени, связанных с работой системного вызова read(). Чтение каждого символа требует обращения к ядру и передачи информации через системный буфер. Более эффективно было бы читать файл блоками.

Пример 3.3. Чтение файла блоками.

```
#include <unistd.h>
#include <fcntl.h>
void exit(int);
main(){
int fd, len;
const int buf_size=4096; // Размер блока
char buffer[buf_size];  // сюда занесем считанный блок
off_t  fend;  // fend=позиция последнего символа в файле
if ((fd=open("text.txt", O_RDONLY, 0220))<0) exit(1);
fend=lseek(fd,0,SEEK_END);  // находим размер файла
lseek(fd,0,SEEK_SET);           // переходим в начало
while (fend>0)        // Пока не достигнут конец
{
   len=read(fd, buffer,buf_size);
   write (1, buffer,len); // Вывод  прочитанного  на терминал
    fend=fend-len; //сколько еще до конца  файла осталось?
}
close(fd);
write (1, "\n",1);
}
```

Рассмотрим приведённый пример. Константа buf_size задаёт предельный размер считываемого блока, а массив buffer, используется как буфер для хранения прочитанного блока текста. Переменная fend содержит количество еще непрочитанных символов. В строке 10 ей присваивается значение, равное позиции последнего символа в файле. Для этого позиция ввода-вывода в строке 10 смещается в конец файла. Затем, в строке 11 она возвращается к его началу.

Собственно чтение происходит в цикле while (строки 12-17). Функция read() считывает (пытается прочитать) в буфер buf_size байт и возвращает реальное количество прочитанных символов. После чего переменная fend уменьшается на это число.

Контрольные вопросы

Какие функции предназначены для работы с файлами?

Каким правам доступа соответствует обозначение 0352?

Что такое файловый дескриптор? Для чего он нужен?

Какое значение возвращает функция open() при удачном завершении?

Зачем нужно проверять код возврата из функций ввода-вывода?

Что означают флаги O_CREAT|O_APPEND при открытии файла?

Если файл открыт для чтения и записи с флагом O_APPEND, можно ли читать данные из произвольного места в файле с помощью функции lseek()?

Можно ли воспользоваться функцией lseek() для изменения данных в произвольном месте в файле?

Что произойдёт, если пользователь создаст файл функцией open("pathname", 0466), запишет в него какую-либо информацию, а затем попробует открыть через файловый менеджер или терминал?

Порядок выполнения работы

Изучите теоретические сведения.

Изучите примеры программ, приведенных в теоретической части.

Модифицируйте примеры 3.1-3.3 для выполнения задания согласно Вашему варианту.

Оформите отчет.

Отчет должен включать:

– титульный лист;

– формулировку цели работы;

– формулировку задания;

– листинги программ;

– результаты выполнения программ;

– выводы, согласованные с целью работы;

– ответы на контрольные вопросы.

Варианты заданий

Вариант 1.

Напишите программы, выполняющие следующие действия.

Программа 1:

1) Создание двух файлов с правами чтения и записи для владельца и чтения для группы и прочих пользователей.

2) Открытие первого файла для записи. При этом, если в нем есть информация, записывать в конец файла. Режим доступа: для владельца – запись, для группы – чтение, для остальных – исполнение.

3) Запись в первый файл аргументов текущей программы (см. ЛР 1);

4) Закрытие файла.

Программа 2:

1) Открытие первого файла для чтения и второго для записи. Если во втором файле есть информация, запись производить в его конец.

2) Копирование последних 20 символов из первого файла во второй.

3) Закрытие обоих файлов.

Вариант 2.

Напишите программы, выполняющие следующие действия:

Программа 1:

1) Открывает текстовый файл для записи. Имя файла указать в качестве первого аргумента программы (см. ЛР 1). Режим доступа: чтение + запись для владельца и чтение для группы. Если файл существует, он должен быть очищен.

2) Пишет в файл строки, введенные пользователем через терминал в бесконечном цикле.

Программа 2:

1) Открывает файл для чтения.

2) Выводит на экран последнее предложение.

3) Закрывает файл.

Вариант 3.

Напишите программы, выполняющие следующие действия.

Программа 1:

1) Открывает текстовый файл для записи. Режим доступа: исполнение и запись для владельца, чтение для группы. Если файл не существует, он должен быть создан, если в файле есть информация, его нужно очистить.

2) Заносит в файл команду(ы) среды исполнения bash:

Если указана опция -p, то выполняется команда ps -aux,

если указана опция -i, то команда info ps.

3) Закрывает файл.

Попробуйте запустить файл на исполнение. Можете ли вы просмотреть его содержимое?

Программа 2:

1) Открывает файл, созданный Программой 1 для чтения. Режим доступа: чтение для владельца и группы.

2) Выводит на экран первое слово в файле.

3) Закрывает файл.

Вариант 4.

Напишите программы, которые выполняют следующие действия:

Программа 1:

1) Создаёт два файла с правами чтения и записи для владельца и чтения и исполнения для группы.

2) Открывает оба файла для дозаписи в конец.

3) Записывает какую-нибудь информацию в оба файла.

4) Закрывает оба файла.

Программа 2:

1) Открывает оба файла для чтения только для владельца и группы.

2) Находит количество совпадений в файлах.

3) Закрывает файлы.

Вариант 5.

Напишите программы, которые выполняют следующие действия:

Программа 1:

1) Открывает текстовый файл для записи в конец. Имя файла указать в качестве первого аргумента программы. Если файл не существует, он должен быть создан, если существует – очищен. Режим доступа – для владельца и группы чтение + запись + исполнение, для остальных – чтение + исполнение.

2) Пишет в файл строку, введенную пользователем.

3) Закрывает файлы.

Программа 2:

1) Открывает файл для чтения только для владельца.

2) выводит из файла случайно выбранное слово.

3) Закрывает файл.

Вариант 6.

Напишите программы, которые выполняют следующие действия:

Программа 1:

Создает пустой файл с правами на запись и исполнение для владельца, чтения и записи для остальных. Имя файла указать в качестве первого аргумента программы.

Открывает созданный файл и пишет в него несколько слов.

Программа 2:

Открывает созданный первой программой файл для чтения.

Выводит каждый третий символ на экран.

Закрывает файл.

Вариант 7.

Напишите программы, которые выполняют следующие действия.

Программа 1:

1) Создаёт текстовый файл. Режим доступа – для владельца и группы запись, для остальных – только чтение.

Если такой файл существует, предложить пользователю изменить имя.

2) Записывает в него информацию.

3) Закрывает файл.

Программа 2:

1) Создает еще один файл для записи с правами доступа чтение+запись для владельца и группы и чтение для остальных;

2) Открывает файл, созданный Программой 1 и новый.

3) Переписывает в него содержимое первого файла, переводя буквы из нижнего регистра в верхний.

Вариант 8.

Напишите программы, которые выполняют следующие действия:

Программа 1:

1) Открывает текстовый файл для чтения и записи. Если файл не существует, его не надо открывать. Файл открывается в режиме чтения, записи и исполнения для владельца.

2) Заносит в файл информацию, введенную с терминала.

3) Закрывает файл.

Программа 2:

1) Открывает файл, созданный ранее.

2) Выписывает все слова, начинающиеся буквы ‘а’.

3) Закрывает файл.

Вариант 9.

Напишите программы, которые выполняют следующие действия:

Программа 1:

1) Создаёт файл с правами доступа на всё для владельца и чтения для всех остальных групп пользователей.

2) Открывает файл и пишет в него содержимое исполняемого файла Программы 2.

3) Закрывает файл.

Программа 2:

1) Открывает созданный файл для чтения и записи.

2) Заменяет все неалфавитные символы точками.

3) Закрывает файл.

Вариант 10.

Напишите программы, которые выполняют следующие действия:

Программа 1:

1) Создаёт два файла с правами доступа на все действия для владельца и чтения группы.

2) Открывает первый файл и пишет в него исходный код Программы 1.

3) Закрывает файл.

Программа 2:

1) Открывает оба файла.

2) Переносит из первого файла во второй все многострочные комментарии.

3) Закрывает файл.

Форма отчетности:

Отчет по работе содержит:

1. Наименование лабораторной работы;
2. Разработанную программу;
3. Результаты её тестирования;
4. Выводы по работе.

Основная литература

1. Лав, Р. Linux. Системное программирование: учебное пособие / Р. Лав; Пер. с англ. - 2-е изд. - СПб. : Питер, 2014.

Дополнительная литература

2. Молчанов, А. Ю. Системное программное обеспечение : учебник для вузов / А. Ю. Молчанов. - Санкт-Петербург : Питер, 2006.

Лабораторная работа 4

Процессы: порождение и управление

Цель работы: Научиться программно запускать процессы, передавать управление новой программе и получать информацию о процессах.

Теоретические сведения

Получение основной информации о процессах

Процесс представляет собой выполняющуюся программу вместе с набором необходимых ей ресурсов. Следует подчеркнуть, понятия программа и процесс не тождественны. Один экземпляр программы может исполняться несколькими процессами, например, будучи запущенным несколько раз. Процесс – это активная единица работы операционной системы.

Для управления процессами операционная система использует два основных типа информационных структур: дескриптор процесса и контекст процесса.

Все дескрипторы процессов собраны в специальный список, называемый таблицей процессов.

Дескриптор процесса представляет собой структуру struct task_struct, которая описана в файле include/linux/sched.h и содержит всю информацию об определенном процессе.

И контекст процесса и дескриптор доступны только из пространства ядра. Однако мы можем получить сведения о запущенных в данный момент процессов, набрав в терминале команду

ps -ax

Опция -a задает все процессы, имеющие терминал, опция
-x позволяет включать процессы, у которых нет управляющего терминала (такие, как демоны, запущенные во время работы)

Если вы хотите ограничить список процессами, запущенными от имени текущего пользователя, добавьте опцию -u:

ps -aux

Система различает процессы по идентификаторам (PID). Каждый идентификатор – это уникальный "номер" процесса в системе. Гарантируется, что никакие два одновременно существующих процесса не имеют одинаковых номеров. Зная PID процесса, вы можете управлять им: приостановить, послать ему сигнал и даже уничтожить.

Программа, запущенная из пространства пользователя, может узнать свой идентификатор и идентификатор своего родителя с помощью функций getpid() и getppid():

```
#include <unistd.h>
int getpid (void);
int getppid (void);
```

Порождение нового процесса

Новые процессы всегда создаются существующими процессами. Существующий процесс называется родительским, а новый процесс — потомком или дочерним процессом. Запуск нового процесса осуществляется функцией fork():

```
#include <unistd.h>
pid_t fork (viod);
```

Как можно понять из названия[2], функция fork() создает копию текущего процесса в новом адресном пространстве.

У родительского и дочернего процессов полностью совпадают код, значения переменных, указатели на используемые ресурсы. Отличаются адресные пространства и идентификаторы.

Ещё одно отличие – значение, полученное в результате вызова fork():

в родительский процесс возвращается PID созданного процесса. Таким образом, родительский процесс получает полный контроль над дочерним;

в дочернем процессе возвращаемое значение равно 0. Благодаря этому, дочерний процесс узнаёт что он – клон.

Если в результате вызова fork() произошел сбой (закончилась свободная оперативная память или заняты все возможные дескрипторы или ещё какая беда приключилась), то будет получено отрицательное значение.

---

[2] fork (англ.) – вилка, раздвоение.

Покажем на примере, как в программном коде описать порождение дочернего процесса.

Пример 4.1. Создание нового процесса

```
#include<unistd.h>
#include<stdio.h>
void exit (int);
int main() {
pid_t clone;  /* переменная clone содержит значение,
                    возвращаемое fork()                    */
clone=fork(); /* попытка "расщепить" процесс              */
/* если попытка не удалась - завершить программу:       */
if (clone <0) {
   printf("Error"); exit (0);
}
/*В дочернем процессе  значение clone равно 0:           */
if (clone= =0) {
   printf ("I am child with PID-%d \n", getpid() );
}
/* в родительском процессе clone равно PID дочернего: */
if (clone>0) {
   printf ("I am parent with PID-%d \n", getpid() );
   printf ("My child's  PID is %d \n" , clone);
}
/* действия, выполняемые всеми процессами:               */
printf ("Good bye!\n");
return 0;
}
```

Переменная clone, объявленная в строке 5 предназначена для хранения идентификатора дочернего процесса. Она инициализируется при вызове функции fork() (строка 7). В результате вызова переменная clone, записанная в старом адресном пространстве получает значение, равное PID дочернего процесса, а её клону, созданному в новом адресном пространстве, присваивается значение 0. Для того, чтобы родительский процесс и его потомок выполняли разные действия, используют конструкцию

```
if (clone==0){
…
}
if (clone>0){
…
}.
```

Все операции, описанные вне этой конструкции, выполняются обоими процессами.

3. Передача управления

Процессы, порождённые вызовом fork() выполняют один и тот же код. Поэтому обычно после создания дочернего процесса, в него передают управление другой программой. Для этого используются функции семейства exec*, объявленные в заголовочном файле unistd.h. Всего в семейство входят 6 функций:

1) int execve(const char * file, char *const argv[], char *const envp[]);
2) int execl(const char *path, const char *argv, ...);
3) int execlp(const char *file, const char *argv, ...);
4) int execle(const char *path, const char *argv, ..., char *const envp[]);
5) int execv(const char *path, char *const argv[]);
6) int execvp(const char *file, char *const argv[]);

Все они выполняют одну работу – заменяют инструкции исходной программы, управляющей процессом на код другой программы.

Когда процесс вызывает одну из функций exec*, то он полностью замещается другой программой, и эта новая программа начинает выполнение собственной функции main(). После вызова exec* возврат в старую программу невозможен. Исключением является случай ошибки: если новую программу запустить не удалось, exec* вернет управление в старую программу и передаст в неё отрицательное значение. Если же запуск новой программы прошел успешно, то старый код будет безвозвратно вытеснен из адресного пространства процесса.

Как мы видим из прототипов, все функции возвращают целое значение. Это кажется бессмысленным, так как после вызова exec* управление безвозвратно передаётся в новую программу. На самом деле, возвращаемое значение используется для сообщения кода ошибки в том случае, если запустить новую программу не удалось. По старой традиции, этом случае в родительский процесс будет возвращено отрицательное число.

Функции семейства exec* несколько отличаются друг от друга по способу обращения к ним и особенностям работы:

Функции, в названии которых есть суффикс 'p' (ecxecvp, execlp) разыскивают заявленную программу с именем file в каталогах, указанных в переменной окружения PATH. Если PATH в окружении не присутствует, они используют путь по умолчанию. В GNU/ Linux по умолчанию используется " : /bin", ": /usr /bin", но в других системах может быть другое значение. (Обратите внимание, что ведущее двоеточие в PATH означает, что сначала поиск осуществляется в текущем каталоге.) Более того, если файл найден и имеет право доступа на исполнение, но не может быть исполнен из-за того, что неизвестен его формат, exec*p() считает, что это сценарий оболочки и запускает оболочку с именем файла в качестве аргумента.

Остальные функции требуют указания полного путевого имени программного файла.

Функции, содержащие суффикс 'v' (execv, execvp, execve) принимают список аргументов загружаемой программы (*argv[]) в виде одного массива, оканчивающегося NULL.

Функции, в имени которых есть суффикс 'l' (execl, execle, execlp) аргументы загружаемой программы принимают списком, по-очереди. Этот список также должен заканчиваться NULL.

Функции, имеющие суффикс 'e' (execle, execve) в качестве дополнительного аргумента принимают указатель на окружение envp.

Рассмотрим подробнее некоторые из этих функций.

Наиболее общий формат имеет функция execve. Её имя совпадает с лежащим в основе семейства системным вызовом. Его прототип имеет вид:

int execve(const char*file, const char *argv[], const char *envp[]);.

Здесь

file – имя бинарного файла, запускающего программу.

argv – указатель на строку аргументов, передаваемых новой программе. Строка должна заканчиваться словом NULL.

envp – указатель на окружение процесса. Строка должна заканчиваться NULL.

Пример 4.2. Фрагмент кода программы, передающей дочерний процесс под управление новой программой progr с помощью execve():

```
…
extern char ** environ;
main (){
char *arguments[]={"progr","1","2",…,NULL};
pid_t child;
…
if ((child=fork())==0){
  execve("./progr",arguments ,environ);
/* До этого фрагмента управление обычно не доходит:   */
   fprintf(stderr, "Не удалось запустить программу progr");
  exit (1);
```

```
}
...
}
```

Показанный выше пример является "скелетом" для построения любой программы, запускающий сторонний код в дочернем процессе. Рассмотрим его основные детали.

В строке

```
extern char ** environ;
```

объявляется указатель на внешнюю (системную) переменную environ, которая используется для хранения переменных среды. Далее в программе можно будет выборочно поменять значения отдельных переменных.

Массив указателей arguments содержит все аргументы вызываемой программы. В соответствии с общепринятыми правилами, нулевой элемент этого массива ссылается на имя программы, далее следуют аргументы программы. Обратите внимание, аргументы 1 и 2 переводятся в строковый формат.

Функция execve() требует предварительной подготовки массива аргументов. В отличие от неё, функция execle() принимает те же аргументы "россыпью":

Пример 4.3. Использование execle()

```
…
extern char ** environ;
main (){
pid_t child;
…
if ((child=fork())==0)  {
   execle("./progr","progr","1","2",NULL ,environ);
   fprintf (stderr, "Не удалось запустить программу progr");
  exit (1);
}
…
}
```

Если требуется запускать из программы shell-скрипты или программы, лежащие в каталогах, указанных в переменной окружения PATH, такие как команды оболочки, то удобнее использовать функции execlp() или execvp(). Рассмотрим пример использования функции execlp():

Пример 4.4. Запуск из дочернего процесса команды оболочки ls -F -a

```
main (){
pid_t child;
if ((child=fork())==0){
   execlp("ls", "ls", "-F","-a", NULL );
   exit (1);
}
...
}
```

Обратите внимание на то, что имя команды "ls" встречается в вызове функции execlp() дважды. Это не ошибка. В первый раз мы указали название запускаемой команды; начиная со второго аргумента, в функцию передаются аргументы программы. При этом первым аргументом (argv[0]) идет имя запускаемой программы. Оканчивается список аргументов пустым указателем NULL.

В результате дочерний процесс выведет на экран список файлов текущего каталога, включая скрытые (опция -a) с информацией об этих файлах (опция -F) .

4. Завершение процесса

Программа может завершиться одним из следующих способов:

1) Естественным образом закончить свою работу и возвратиться из main().

2) Если в процессе работы возникла ситуация, способная привести к аварии или препятствующая выполнению программой своих обязанностей, то можно организовать досрочный выход с помощью функций exit() или abort():

void exit (int status) – завершает работу программы и передаёт системе значение status – код возникшей ситуации. Принято указывать значение status равное 0 при отсутствии ошибки и положительное целое значение в противном случае.

viod abort(void) – вызывает аварийное завершение процесса. При этом буферы не выгружаются и потоки не закрываются. Вместо неё предпочтительней использовать exit().

Функции exit() и abort() обеспечивают завершение программы из любой её точки, а не только из main().

3) Процесс может завершиться по сигналу извне. Если известен pid процесса, то его можно завершить, послав один из сигналов SIGINT, SIGTERM, SIGKILL с помощью функции kill():

#include <unistd.h>

int kill(pid_t pid, int sig);

Здесь

pid – идентификатор процесса, которому отправляется сигнал;

sig – номер отправляемого сигнала.

Если в переменной child хранится идентификатор дочернего процесса, то запрос на его прерывание из родительского будет выглядеть так:

kill(child, SIGINT);

5. Ожидание завершения дочерних процессов

Если исходная программа запустила дочерний процесс, то далее они будут выполняться независимо друг от друга. В некоторых случаях необходимо, чтобы родительский процесс дожидался завершения дочернего. Также родительскому процессу бывает необходимо знать, как именно завершился дочерний процесс – нормально, по ошибке или в результате принятого сигнала.

В заголовочном файле <wait.h> объявлены функции wait() и waitpid():

#include <wait.h>

pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int options);

Функция wait() ждет завершения любого порожденного процесса; сведения о том, как он завершился, возвращаются в переменную * status.

Функция waitpid() заставляет родительский процесс ожидать завершения потомка с заданным значением pid.

При вызове функции семейства wait*() в неё передаётся указатель на переменную status, куда в результате вызова будет записана информация о причине завершения процесса.

Формат возвращенного состояния довольно сложный, и для того, чтобы сделать его осмысленным, существует набор макросов, каждый из которых принимает возвращаемое состояние в качестве единственного параметра.

WIFEXITED(status) Возвращает true, если процесс завершился нормально. Процесс завершается нормально, когда его функция main () выходит из программы посредством return 0; или вызова exit ().

WEXITSTATUS (status) Если WIFEXITED истинно, то WEXITSTATUS (status) возвращает код возврата процесса.

WIFSIGNALED(status) Возвращает true, если процесс был прерван сигналом.

WTERMSIG (status) возвращает номер сигнала, прервавшего процесс.

Аргумент options управляет поведением вызова. Этот параметр должен быть равен либо 0, либо побитовым ИЛИ одного или более из следующих флагов:

WHOHANG Если ни один порожденный процесс не завершился, вернуться немедленно.

WUNTRACED Вернуть сведения о порожденном процессе, который остановился, но еще не завершился.

Первый аргумент, pid, обычно представляет собой процесс, завершение которого ожидается. Однако можно использовать и другие значения pid, а именно:

pid < -1 – ожидать завершения любого порожденного процесса с ID группы процесса, равной абсолютному значению pid.

pid = -1 – ожидать прерывания любого дочернего процесса.

pid = 0 Ожидать завершения дочернего из той же группы процессов, что и текущий.

Пример 4.5. Ожидание завершения дочернего процесса

```
#include<unistd.h>
#include <wait.h>
#include<stdio.h>
void exit (int);
main(){
int exit_status;
pid_t clon=-1; // PID дочернего процесса
if ((clon=fork())<0)
{
  printf ("Не удалось создать дочерний процесс \n");
  exit(1);
}
if (clon==0) {   /* Дочерний процесс  */
  usleep(500);  // пауза в 500 мс
  if( ( execlp("ps","ps","-f",NULL ))<0)
 {     /* До этого места управление обычно не доходит  */
printf ("Запуск программы ps не удался  \n");
exit(1);
}
}
/*
 Код ниже выполняется лишь родительским процессом:
*/
/* Ожидание завершения дочернего процесса:              */
 waitpid(clon,&exit_status,0);
/* Обработка статуса завершения дочернего процесса: */
if (WIFEXITED(exit_status))
{
  printf("Дочерний процесс завершился благополучно");
  printf ("код возврата %d \n",WEXITSTATUS(exit_status));
}
if (WIFSIGNALED(exit_status))
{
  printf("Дочерний процесс прерван сигналом");
  printf(" с номером  %d \n", WTERMSIG(exit_status));
 }
 printf ("Родительский процесс завершен\n");
}
```

В строке 8 приведенной в примере программы, выполняется попытка создать копию работающего процесса. Если это не удалось, то системный вызов fork() вернет отрицательное значение и программу нужно будет завершить.

Если создание дочернего процесса прошло успешно, то в дочерний процесс вызывается команда среды исполнения ps с опцией –f. Функция waitpid() приостанавливает дочерний процесс до завершения дочернего и получает код завершения, передаваемый по адресу переменной &exit_status. Далее в строках 27-35 производится обработка значения этой переменной.

Контрольные вопросы

Какую функцию нужно использовать, чтобы получить PID текущего процесса?
Какую работу выполняет функция fork()?
Что общего у родительского процесса и дочернего, порожденного вызовом fork()?
Чем отличаются родительский и дочерний процессы?
Можно ли гарантировать, что родительский и дочерний процессы будут выполнять свои операции в определённом порядке? Ответ обоснуйте.
Какие функции используются для загрузки нового кода в текущий процесс?
Может ли программа, вызванная exec*() обращаться к переменным, объявленным ранее в родительском процессе?
В чем различие между функциями wait() и waitpid()?

Порядок выполнения работы

Изучите теоретические сведения.
Изучите примеры программ, приведенных в теоретической части.
Напишите программу, которая 3 раза вызывает функцию fork() и затем выводит сообщение, содержащее свой PID и PID своего родителя. Сколько сообщений было выведено на экран? Объясните полученные результаты.
Модифицируйте предыдущую программу так, чтобы было выведено ровно 3 сообщения.
Напишите программу, в которой открывается текстовый файл test.txt, затем создаётся 2 дочерних процесса без передачи им стороннего управления. Действия, выполняемые процессами, выберите из таблицы 4.1 согласно Вашему варианту.
Родительский процесс должен дожидаться окончания дочерних, и только после этого завершаться. Что будет, если родительский процесс завершиться преждевременно?
Напишите программу, запускающую дочерние процессы с вызовом в них новых программ согласно Вашему варианту (см табл. 4.2.) в следующем порядке:
1) Вызов Программы 1;
2) Вызов команды ps -au;
3) Вызов Программы 2;
4) Вызов команды ps -au;
5) Ожидание завершения всех дочерних процессов;
6) Вызов команды ps -au.
Оформите отчет.

Отчет должен включать:
– титульный лист;
– формулировку цели работы;
– формулировку задания;
– листинги программ;
– результаты выполнения заданий;
– выводы, согласованные с целью работы;
– ответы на контрольные вопросы.

Варианты для задания 4

№ вар	Дочерний процесс 1	Дочерний процесс 2
1	Пишет в файл английский алфавит.	Считывает файл по одной букве и выводит на экран.
2	Считывает из файла по одной букве и выводит их на экран.	"Шагает" по файлу на 2 символа назад и пишет '01' в текущую позицию.

3	Пишет в файл букву 'a' 10 раз.	Пишет в файл букву 'b' 10 раз.
4	Считывает и выводит на экран текущий символ.	Перемещается на шаг назад и пишет '.' в текущую позицию в файле.
5	"Шагает" по файлу на 1 символ назад. Выводит текущую позицию в файле.	Пишет в файл английский алфавит.
6	"Прыгает" в случайное место в файле, не совпадающее с последним символом.	Сравнивает две соседние буквы. Если они располо-жены в алфавитном порядке, то меняет их местами.
7	Выводит текущее слово из файла на экран.	Заносит в файл 10 букв, сгенерированных случайным образом.
8	Считывает по одной букве из файла в обратном порядке и выводит на экран.	Считывает по одной букве из файла в прямом порядке и выводит их на экран.
9	Считывает последний символ и заменяет его следующим по алфавиту.	Записывает в файл символы, введенные пользователем с клавиатуры.
10	Считывает из файла по одной букве, пока он не закончится. Выводит результаты на экран.	

Таблица 4.2.
Варианты для задания 5

№ вар	Программа 1	Программа 2
1	Программа 2 из ЛР 3	cat файл (файл - имя результирующего файла изЛР 3)
2	info write > файл (здесь файл - имя файла, открываемого в ЛР 3.2)	Программа 2 из ЛР 3
3	ls -a > файл (здесь файл - имя файла, открываемого в ЛР 3.2)	Программа 2 из ЛР 3
4	Программа 1 из ЛР 3	comm файл1 файл2 (здесь файл1 и файл2- файлы, открывемые в ЛР 3.1)
5	Программа 1 из ЛР 3	tac файл (здесь файл - имя файла, открываемого в ЛР 3.1)
6	Программа 1 из ЛР 3	paste файл1 файл2 (здесь файл1 файл2- файлы, открываемые в ЛР 3.1)
7	Программа 2 из ЛР 3	paste файл1 файл2 (здесь файл1 файл2 - файлы, открываемые в ЛР 3.1)
8	Программа 1 из ЛР 3	cat файл (здесь файл - имя результирующего файла из ЛР 3)
9	Программа 1 из ЛР 3	sort файл (здесь файл - имя результирующего файла из ЛР 3)

10	Программа 2 из ЛР 3	comm. файл1 файл2 (здесь файл1 файл2 - файлы, открываемые в ЛР 3.1)

Форма отчетности:

Отчет по работе содержит:

1. Наименование лабораторной работы;
2. Разработанную программу;
3. Результаты её тестирования;
4. Выводы по работе.

Основная литература

1. Лав, Р. Linux. Системное программирование: учебное пособие / Р. Лав; Пер. с англ. - 2-е изд. - СПб. : Питер, 2014.

Дополнительная литература

2. Молчанов, А. Ю. Системное программное обеспечение : учебник для вузов / А. Ю. Молчанов. - Санкт-Петербург : Питер, 2006.

Лабораторная работа 5.

Сигналы

Цель работы: Ознакомиться с понятием сигнала в системе. Освоить способы перехвата и обработки сигналов.

Теоретические сведения

1. Отправка сигнала процессу

Отправить сигнал другому процессу очень легко. Для этого достаточно знать PID этого процесса. Для отправки сигналов используют функцию kill:

```
#include <unistd.h>
int kill(pid_t pid, int SIGNAL); ,
```

где

pid – идентификатор процесса, которому предназначен сигнал,

SIGNAL – номер или имя посылаемого сигнала.

В качестве примера рассмотрим программу, которая порождает дочерний процесс и отправляет ему три сигнала с интервалом в 2 секунды.

Пример 5.1. Отправка сигналов дочернему процессу

```
#include <unistd.h>
#include <signal.h>
main(){
  pid_t child;
   if((child=fork())<0) return;
   if(child==0)      {   // код  дочернего процесса
    while(1){
        write(2,".",1); usleep(10000);  //  печатаем точки
     }
   }
   sleep(2);
   kill(child, SIGSTOP);  // Приостановить выполнение
   write(1,"Пауза\n",12);
   sleep(2);
   kill(child, SIGCONT);  // Продолжить выполнение
   sleep(2);
   kill(child, SIGINT);      // Прервать процесс
}
```

Если мы запустим эту программу на выполнение, то увидим что она сначала пишет точки в терминале, затем приостанавливается на 2 секунды, затем снова принимается писать точки и ещё через две секунды завершается.

2. Перехват сигнала

Структура sigaction

Структура sigaction содержит все сведения, необходимые программе для перехвата сигнала. В частности, в неё записывается адрес функции-обработчика.

Основные поля этой структуры:

sa_mask Задает маску сигналов, которые должны дополнительно блокироваться во время работы функции-обработчика.

sa_flags Содержит набор флагов, которые могут влиять на поведение процесса при обработке сигнала.

*sa_handler Задает указатель на функцию обработки или тип действий процесса, связанный с сигналом sig. Может быть равен SIG_DFL , SIG_IGN или имени функции–обработчика.

Маска сигналов sa_mask блокирует указанные в ней сигналы только во время обработки принятого сигнала. После возврата в основную функцию программы, прием сигналов возобновляется. Для управления маской используются специальные функции, представленные в таблице 5.1.

Таблица 5.1.

Функции, управляющие маской сигнала

Функция	Назначение
sigemptyset (sigset_t *set)	Инициализирует пустой набор сигналов
sigfillset (sigset_t *set)	Инициализирует полный набор сигналов
sigaddset(sigset_t *set , int sig)	Добавляет сигнал в набор
sigdelset (sigset_t *set , int sig)	Удаляет сигнал из набора
sigismember (sigset_t *set , int sig)	Проверяет наличие сигнала в наборе

Поле sa_flags содержит один или более флагов, объединенных битовой операцией "ИЛИ":

SA_NOCLDSTOP – не принимать сигнал о приостановке дочернего процесса. Обычно сигнал SIGCHLD генерируется, когда один из потомков процесса прерван или приостановлен. Если флаг SA_NOCLDSTOP указан для сигнала SIGCHLD, то сигнал генерируется лишь в случае прерывания дочернего процесса; приостановка дочернего процесса не приводит к генерации каких-либо сигналов.

SA_RESTHAND – восстановить действие сигнала по умолчанию после обработки. Когда присылается сигнал, обработчик сбрасывается в SIGDFL.

SA_RESTART – возобновить выполнение прерванного системного вызова. Когда сигнал посылается процессу во время выполнения медленного системного вызова, системный вызов перезапускается после возврата управления из обработчика.

SA_NODEFER – эмуляция простых (ненадежных) сигналов.

Функция sigaction()

Функция sigaction() организует перехват сигнала. Она имеет следующий прототип:

#include<signal.h>

int sigaction(int sig, struct sigaction *act, struct sigaction *oldact);.

Здесь

sig –номер сигнала;

*act – указатель на структуру sigaction, описывающую новые правила обработки сигналов;

*oldact – указатель структуру, описывающую текущий обработчик сигнала. Используется для того, чтобы иметь возможность вернуться к предыдущему способу обработки сигналов. Если в этом нет необходимости, значение *oldact устанавливается равным NULL.

Пример 5.2. Заполнение структуры sigaction и назначение обработчика для сигналов SIGINT, SIGTSTP и SIGALRM.

```
#include <signal.h>
/*   Объявление обработчика сигналов:              */
void handler (int sig);
main () {
/*   Объявление структуры sigaction:                */
struct sigaction sig_act;
/*  Заполнение полей структуры:
Предварительная очистка маски;
Добавление сигнала SIGCHLD в маску;
Установка флагов;
Назначение обработчика сигналам.
*/
sigemptyset (&sig_act.sa_mask);                      //1
sigaddset (&sig_act.sa_mask, SIGCHLD);               //2
sig_act.sa_flags=SA_RESTART| SA_NOCLDSTOP; //3
sig_act.sa_handler=&handler;                         //4
/* Организация перехвата сигналов:                */
sigaction (SIGINT, &sig_act, NULL);
sigaction (SIGALRM, &sig_act, NULL);
sigaction (SIGTSTP, SIG_IGN, NULL);
…
```

3. Обработка сигнала

После того как сигнал пойман, программа должна прервать обычный ход выполнения инструкций и переключиться на его обработку. Для этого пишут функцию-обработчик сигнала.

Функция-обработчик – это обычная функция пользователя, в которую передаётся управление при получении сигнала. Вы можете использовать в программе несколько функций для обработки сигналов, но все они должны иметь один и тот же прототип:

```
void имя_обработчика (int SIGNAL);
```

Функция-обработчик принимает один-единственный аргумент – значение сигнала и не возвращает никаких значений в вызвавшую её программу.

Рассмотрим пример программы, которая перехватывает нажатие пользователем комбинаций клавиш Ctrl+C и Ctrl+Z, что соответствует отправке сигналов SIGINT и SIGTSTP. Наша программа будет подсчитывать количество поданных сигналов SIGTSTP и выводить соответствующее сообщение. После сигнала SIGINT будет выведено соответствующее сообщение и программа прекратит свою работу.

Пример 5.3. Обработка сигналов SIGINT и SIGTSTP

```
#include <stdio.h>
#include <signal.h>
void exit (int);
void handler (int sig);   // Прототип обработчика
main () {
  /* Подготавливаем структуру act для обработки сигналов*/
   struct sigaction sig_act;
   sig_act.sa_flags=SA_RESTART;
   sig_act.sa_handler=&handler;
   /* Организуем перехват сигналов:                    */
   sigaction (SIGINT, &sig_act,NULL);
```

```
    sigaction (SIGTSTP, &sig_act,NULL);
    /* Вне обработки сигнала программа может что-либо    делать, например,
осуществлять ввод символа:                */
    int c;
    while (1) {
      c=getchar();
     }
    printf( "\n Good bye! \n");
  }
  /*************************************************/
  /* Функция-обработчик сигнала                                    */
  void handler (int sig) {
    static int  count=0;   /*  Счётчик числа сигналов              */
    switch (sig) {
      case SIGTSTP:
          printf ("Сигнал STOP № %d\n", count);
          count++; break;
      case SIGINT:
          printf("Конец работы \n"); exit(0);
      default: break;
    }
  }
```

Для подсчета числа сигналов мы объявили в функции-обработчике статическую переменную count. Спецификатор static сигнализирует компилятору о том, что переменная должна сохранять свои значения между вызовами функции.

5. Режим реального времени

Будильник alarm()

В программах, работающих в режиме реального времени часто используются таймеры – функции, отсчитывающие заданный интервал времени.

Простейший таймер запускается с помощью функции alarm():

```
#include <unistd.h>
unsigned int alarm(unsigned int sec);
```

Данная функция отсчитывает sec секунд, после чего подаёт сигнал SIGALRM.

Если в аргументе sec указать число 0, таймер отключится.

Напишем программу, которая запрашивает код доступа у пользователя. Пользователю предоставляется неограниченное число попыток в течение промежутка времени, равного 10 секундам. Если пользователь за это время не вводит верный код, программа прекращает работу:

Пример 5.4. Использование будильника alarm():

```
#include <unistd.h>
#include <stdio.h>
main() {
  int key;
  alarm(10);  // завести" будильник
  printf("Введите код доступа: \n");
  while(1){
    scanf("%d",&key);
    if (key==1234){     // если код введен верно,
         alarm(0);        // остановить таймер
         break;               // и выйти из цикла
     }
      printf("Код неверен. Повторите ввод \n");
  }
  printf("Добро пожаловать в программу ");
```

}

В 5 строке приведённого здесь кода устанавливается таймер на 10 секунд. Затем организуется бесконечный цикл (while(1)) ожидания ввода. Если пользователь так и не введет правильный код, в нашем случае – 1234, то по истечении 10 секунд программа сама завершится. Функция alarm() по истечении времени посылает сигнал SIGALRM. Действием по-умолчанию для него является немедленное завершение работы программы. Если нужно, чтобы при получении сигнала, программа выполняла какие-нибудь другие действия, нужно писать обработчик сигнала.

Во многих современных операционных системах ведутся журналы событий. Это текстовые файлы, куда записываются сообщения о событиях, которые система считает важными. Видоизменим пример 5.4 так, чтобы при срабатывании таймера в файл event.txt записывалось сообщение о несанкционированной попытке доступа к программе:

Пример 5.5. Обработка сигнала SIGALRM

```
#include <stdio.h>
#include <unistd.h>
#include<fcntl.h>
#include <signal.h>
#include<sys/time.h>
void exit (int);
void sig_handler (int sig);   // обработчик сигнала
main() {
   struct sigaction sig_act;
   sigemptyset(&sig_act.sa_mask);
   sig_act.sa_flags=SA_RESTART;
   sig_act.sa_handler=&sig_handler;
   sigaction(SIGALRM,&sig_act,NULL);
   int key;
   alarm(10);
   printf("Введите код доступа: \n");
   while(1){
      scanf("%d", &key);
      if (key==1234){  // если код введен верно,
         alarm(0);        // остановить таймер
          break;           // и выйти из цикла
      }
      printf("Код неверен. Повторите ввод\n");
      }
   printf("Добро пожаловать в программу \n");
   }
/********** Обработчик сигнала**********/
void sig_handler(int sig){
   time_t now;
   time(&now);
   int fd;
   fd=open("event.txt", O_WRONLY|O_APPEND,0600);
   write(fd,"попытка взлома:  ",29);
   write(fd,ctime(&now),31);
   close(fd);
    exit(1);
}
```

Если сравнить это пример с предыдущим, то можно заметить, что основная часть программы, отвечающая за её непосредственную работу, не изменилась. В нашем новом примере это строки 13-25.

Организация задержки выполнения программы

Будильник alarm() не приостанавливает работу программы. Он просто выжидает положенное время, чтобы затем подать сигнал. Для того, чтобы приостановить выполнение программы применяют функции sleep(), usleep() и pause(), объявленные в unistd.h:

sleep (unsigned int s) – приостанавливает работу процесса до истечения заданного времени или пока не будет получен сигнал, который нельзя проигнорировать;

usleep (unsigned int ms) – приостанавливает работу на интервал времени, измеряемый с точностью до микросекунды, либо пока не будет доставлен сигнал;

pause() – приостанавливает работу до тех пор, пока не будет доставлен какой-либо сигнал.

Примечание. Совместное использование sleep() и alarm() является крайне нежелательным.

Интервальные таймеры

Альтернативой alarm() является система интервальных таймеров. Она имеет более высокую точность при измерении времени и допускает использование не одного, а трех таймеров для каждого процесса. И это еще не всё. Для каждого из этих таймеров допустимо использовать две установки: установка режима единичного срабатывания, как в alarm(), и установка режима периодичного срабатывания таймера.

С каждым процессом ассоциированы три таймера:

ITIMER_REAL. Отслеживает реальное время работы.

Генерирует сигнал SIGALRM. Несовместим с системным вызовом alarm() и функцией sleep ().

ITIMER_VIRTUAL. Подсчитывает время только при исполнении процесса в режиме пользователя — не учитывая системные вызовы, которые производит процесс. Если процесс заблокирован во время ввода/вывода, например, на диск или на терминал, таймер приостанавливается.

Генерирует сигнал SIGVTALRM.

ITIMER_PROF. Подсчитывает время только при выполнении процесса, включая время, за которое ядро посылает исполнительные системные вызовы от имени процесса, и не включая время, потраченное на прерывание процесса по инициативе самого процесса. Он действует все время, пока выполняется процесс, даже если операционная система делает что-нибудь для процесса (вроде ввода/вывода).

Генерирует сигнал SIGPROF.

Объявление и настройка таймера производится структурой struct itimerval:

```
#include <sys/time.h>
struct itimerval
 {
   struct timeval it_interval;
   struct timeval it__value;
};
```

Член it_value показывает задержку по времени до отправления первого сигнала.

Член it_interval определяет время между сигналами. Каждый раз при истечении таймера это значение присваивается переменной itvalue.

Как мы можем видеть, поля структуры itimerval имеют тип другой структуры – timeval. Последняя представляет собой способ обозначения времени в формате sec:usec (секунды:микросекунды) и выглядит следующим образом.

```
#include <sys/time.h>
struct timeval
{
   long tv_sec;  // число секунд
   long tv_usec;  //число микросекунд
};
```

Для взаимодействия с интервальными таймерами предусмотрены два системных вызова. Оба принимают аргумент which, указывающий обрабатываемый таймер.

#include <sys/time.h>

int setitimer(int which, struct itimerval *new, struct itimerval *old);

int getitimer (int which, struct itimerval *val) ; .

Здесь

setitimer() устанавливает для данного таймера значение в new. Если имеется указатель old, функция заполняет ее текущим значением таймера. Если нет необходимости сохранять текущие значения настроек таймера, в качестве третьего аргумента можно задать пустой указатель NULL.

getitimer() позволяет считать информацию о настройках таймера which. Для этого он заполняет struct itimerval, на которую указывает val, текущими установками данного таймера.

Значения таймеров уменьшаются от величины it_value до нуля, после чего подаётся сигнал и в поле it_value загружается значение, записанное в it_interval. Поле it_interval содержит константные значения. Ввод нулевого значения для itinterval отключает таймер после запуска. Если установить все поля it_value равными 0, то таймер вообще не запустится.

Рассмотрим основные этапы работы с таймерами на следующем примере. Программа запускает таймер и, после пятого срабатывания, изменяет его настройки.

Пример 5.6. Запуск таймера

```
#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>
#include<signal.h>
int tick=0;  /*Счетчик числа срабатываний таймера        /*
void exit(int);
 /* Обработчик сигнала                                          */
void handler(int sig)
  {
      printf("Получен сигнал таймера %d \n",tick ++);
}
main() {
/* Организуем перехват сигналов таймера:                    */
  struct sigaction act;
  sigemptyset(&act.sa_mask);
  act.sa_flags=SA_RESTART;
  act.sa_handler=&handler;
  sigaction(SIGALRM,  &act, NULL);
   /* Объявляем структуру itimerval                          */
   struct itimerval my_timer;
  /* Предварительно очищаем поля структуры,
     чтобы не было сбоя                                       */
   timerclear(&my_timer.it_interval);
   timerclear(&my_timer.it_value);
   /    * Производим  настройку таймера                      */
   my_timer.it_interval.tv_sec=1;      //Период 1 секунда
   my_timer.it_value.tv_usec=10;      // Задержка 10 мкс
   /* Запуск таймера                                             */
   setitimer(ITIMER_REAL, &my_timer,NULL);
   printf("СТАРТ \n");
   while(tick<5) ;  /* Пустой цикл */
   /* Изменяем настойки таймера и перезапускаем его     */
   my_timer.it_interval.tv_sec=5;
```

```
    my_timer.it_value.tv_sec=1;
    setitimer(ITIMER_REAL, &my_timer,NULL);
    while(i<10) ;  /* Пустой цикл */
}
```

Контрольные вопросы

Каким образом родительский процесс отправляет сигнал дочернему?

Среди перечисленных сигналов, укажите управляющие заданиями: SIGINT, SIGFPE, SIGSTOP, SIGKILL, SIGCHILD.

Какие варианты реакции на сигнал предусмотрены системой?

Какие два сигнала невозможно перехватить или проигнорировать?

Для чего нужна структура sigaction?

Может ли функция-обработчик сигнала возвращать целое значение?

Как Вы думаете, почему не рекомендуется совместное использование функций alarm() и sleep()?

Какие виды таймеров используются в системе?

Порядок выполнения работы

Изучите теоретические сведения.

Изучите примеры программ, приведенных в теоретической части.

Выполните своё задания согласно варианту.

Получите распечатки примеров работы программы.

Оформите отчет.

Отчет должен содержать

– титульный лист;

– формулировку цели работы;

– формулировку задания;

– листинги программ;

– результаты выполнения заданий;

– выводы, согласованные с целью работы;

– ответы на контрольные вопросы.

Варианты заданий

Вариант 1

Напишите программу, тренирующую у пользователя скорость печати. Программа выводит на экран случайную букву алфавита, а пользователь должен нажать соответствующую ей клавишу за ограниченное время. Если пользователь не успел нажать клавишу в отведённый ему промежуток времени, программа предлагает новое задание.

Вариант 2

Организуйте попеременную запись данных в текстовый файл двумя родственными процессами с помощью сигналов. Каждый процесс пишет в файл некий текст с указанием отправителя, затем посылает другому процессу сигнал и приостанавливается до получения ответного сигнала.

Далее ситуация несколько раз повторяется. Текст должен быть осмысленным, сообщения в файле от обоих процессов должны чередоваться.

Вариант 3

Напишите программу-тест по арифметике. Программа предлагает пользователю бесконечный ряд примеров по арифметике и принимает ответы. Примеры генерируются с помощью датчика случайных чисел. Если пользователь хочет "подсмотреть" ответ, то он генерирует сигнал SIGINT. Программа работает ровно 2 минуты и завершается. По окончании выводится количество правильных ответов.

Вариант 4.

Организуйте взаимодействие двух родственных процессов посредством сигналов. Процессы имитируют взаимодействие водителя и навигатора. Навигатор (родительский

процесс) генерирует случайный "маршрут". Затем раз в секунду подает процессу-водителю один из трех сигналов. Номера сигналов выберите сами. Значения сигналов следующие:

Сигнал 1 - прямо; Сигнал 2 - направо; Сигнал 3 - налево.

Водитель (дочерний процесс) принимает от дочернего сигналы, определяет текущее направление движения и пишет его в файл report.

Вариант 5

Напишите модификацию игры "Виселица". Программа выбирает из заранее подготовленного текстового файла случайное слово и предлагает пользователю угадать его по буквам. Время угадывание ограничено N секундами. Пользователь может поменять слово и перезапустить таймер, нажав сочетание клавиш Ctrl+Z.

Вариант 6

Организуйте поочередный доступ к файлу для двух родственных процессов. Первый процесс пишет в файл два случайных числа, после чего посылает сигнал второму процессу. Второй процесс, получив сигнал, считывает числа и находит максимальное. Затем ситуация повторяется ещё 3 раза.

Вариант 7

Организуйте взаимодействие двух родственных процессов посредством сигналов. Процессы имитируют часы с кукушкой. "Часы" запускают "Кукушку", затем каждые 500 мс пишут "тик-так". "Кукушка" каждые 3 секунды отмечает новый час. По достижении полуночи оба процесса завершаются.

Вариант 8

Смоделируйте ситуацию попытки подключения к удаленному серверу. Процесс-родитель организует диалог с пользователем и запускает дочерний процесс. Дочерний процесс имитирует попытки подключения. Для этого он генерирует случайное число от 0 до 10. Если это число меньше 3, то посылает родителю Сигнал 1, если больше или равно - Сигнал 2 и завершается. Родительский процесс в ответ на первый сигнал пишет "Связь установлена" и завершается , а в ответ на сигнал 2 - "Ошибка подключения" и предлагает повторить попытку.

Вариант 9

Напишите программу-игру с таймером. В начале пользователь указывает время, на которое следует завести таймер. В процессе игры пользователь может посмотреть текущее время, нажав сочетание клавиш Ctrl+Z. Предусмотреть перехват и обработку сигнала таймера и вывод соответствующих результатов.

Вариант 10

Напишите программу-игру с таймером. Каждые N секунд игра прерывается сообщением с текущим временем. Одновременно с этим в файл report программа сбрасывает сообщение о последнем игровом действии пользователя.

Форма отчетности:

Отчет по работе содержит:

1. Наименование лабораторной работы;
2. Разработанную программу;
3. Результаты её тестирования;
4. Выводы по работе.

Основная литература

1. Лав, Р. Linux. Системное программирование: учебное пособие / Р. Лав; Пер. с англ. - 2-е изд. - СПб. : Питер, 2014.

Дополнительная литература

2. Молчанов, А. Ю. Системное программное обеспечение : учебник для вузов / А. Ю. Молчанов. - Санкт-Петербург : Питер, 2006.

Лабораторная работа № 6

Неименованные каналы

Цель работы: Изучить особенности функционирования неименованных каналов. Научиться организовывать передачу данных от процесса к процессу. Освоить применение программных каналов и программ-фильтров.

Теоретические сведения

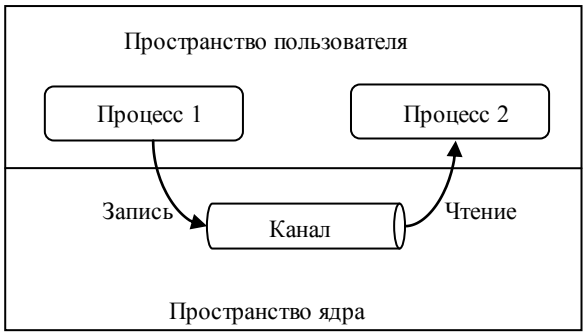
1. Общие сведения о каналах

Канал — это однонаправленный поток байтов, используемый для передачи информации между процессами.

Канал представляет собой буфер в памяти ядра операционной системы (рисунок 1). Когда процесс создает канал, ядро устанавливает два файловых дескриптора для пользования этим каналом. Один такой дескриптор используется, чтобы писать в канал, в то время как другой применяется для чтения данных из канала.

Рисунок 6.1— взаимодействие процессов через канал

В Linux ёмкость канала записана в системную константу PIPE_BUF, объявленную в limits.h. Мы можем узнать эту величину следующим образом:




```
#include<stdio.h>
#include<limits.h>
main() {
printf("%d ",PIPE_BUF);
}
```

Данные, направленные в канал процессом-отправителем, не обязательно должны быть немедленно прочитаны процессом-получателем, но могут накапливаться в канале. Когда ёмкость канала будет исчерпана, запись в него становится невозможной.

2. Создание каналов

Каналы создаются системным вызовом pipe() и одноимённой функцией:

```
#include<unistd.h>
int pipe (int fd[2]);
```

Здесь fd[2] — массив дескрипторов для чтения и записи. Значение fd[0] является читаемым концом канала, а fd[1] — записываемым концом. Удобным способом запоминания является то, что читаемый конец использует индекс 0, аналогичный дескриптору стандартного ввода 0, а записываемый конец использует индекс 1, аналогичный дескриптору стандартного вывода 1.

Пример 6.1. Основные этапы работы с каналом

```
 #include <unistd.h>
main(){
  int fd[2]; // Объявление дескрипторов канала
….
  if (pipe(fd)<0) printf("Не удалось открыть канал");
…
    close (fd[0]); /* Чтобы писать в канал, нужно закрыть  дескриптор на чтение */
…
}
```

Сам вызов pipe () возвращает 0 при успешном выполнении и -1, если была ошибка. После работы канал закрывается с помощью вызова close ().

3. Передача данных по каналу

Согласно общей концепции представления данных в Linux, канал реализуется как файл в виртуальной файловой системе. Поэтому операции чтения и записи в канал

выполняются теми же системными вызовами, что и для обычных файлов. То есть read() и write(). Для передачи данных один из процессов закрывает дескриптор канала, предназначенный для чтения, а другой — для записи. Напишем две программы, которые взаимодействуют друг с другом посредством канала. Первая программа (dialog) будет отвечать за взаимодействие с пользователем и отправлять в канал данные. Вторая (analysis) – обрабатывать поступающие данные.

Пример 6.2. Взаимодействие процессов через канал

Код родительского процесса (dialog.c)

```
#include <unistd.h>
#include <stdio.h>
#include<string.h>
#include<wait.h>
void exit(int);
main () {
int fd[2];        // дескрипторы концов канала
char sfd [10];
pid_t child ;
/* Создаем канал : */
pipe(fd);
/* подготавливаем дескриптор для чтения из канала
    для передачи его в дочерний процесс                    */
sprintf (sfd, "%d",  fd[0]);
child=fork();
if (0==child) {
/*Закрываем записывающий конец канала                    */
  close (point[1]);
/* Запуск дочернего процесса и передача ему
    дескриптора читаемого конца                            */
 execl("./analysis"," analysis", sfd, NULL);
 exit(0);
}
/* Взаимодействие с пользователем                         */
char login[100];
int length;
close(fd[0]);
printf("Введите логин \n");
scanf("%s",login);
length=strlen(login);
login[length]='\0';
/* Записываем данные в канал                              */
write(fd[1],login,length+1);
/* Ожидаем завершения дочернего процесса                  */
int exit_status;
wait(&exit_status);
}
```

Код дочернего процесса (analysis.c)

```
##include <unistd.h>
#include <stdio.h>
#include <string.h>
main (int argc, char **argv) {
char* users[]={"root", "usver", "hacker", NULL};
char login[100];
int fd=atoi(argv[1]);
/* Чтение из канала                                       */
```

```
read (fd, login, 100);
/* Обработка данных                                    */
int find=0,i=0;
while(users[i]){
  if(!(strcmp(log[i],login) )){
    printf("Есть такой \n");  find=1;  break;
  }
 i++;
 }
 if(find==0)   printf("Неизвестный логин\n" );
}
```

Откомпилируем программы:

cc -o dialog dialog.c

cc -o analysis analysis.c

Теперь запустим программу-родителя. Родитель, в свою очередь, запускает дочерний процесс и передает ему через канал данные для обработки. Как можно видеть на рисунке 6.1 запрос имени пользователя корректно обрабатывается.

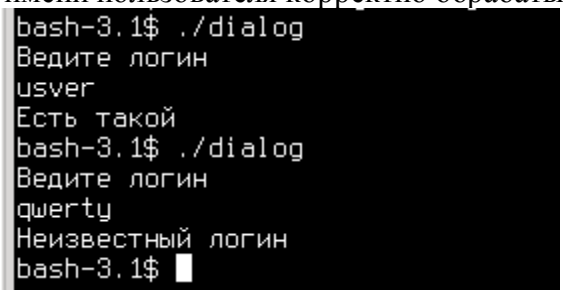


Рисунок 6.1. – Результаты работы программ

Вначале мы подготавливаем массив fd[2] для дескрипторов канала (строка 7) программы dialog. Также нужно позаботится о том, чтобы сообщить дочернему процессу дескриптор читаемого конца fd[0]. Единственный возможный здесь способ — передать его в качестве аргумента программы. Но для этого нужно перевести числовое представление fd[0] в строковое. Здесь используется функция sprintf (строка 14). Функция sprintf() по синтаксису подобна функции fprintf(), но записывает результат не в файл, а в строку.

Обратим внимание на строки 35-36 родительского процесса. Здесь организовано ожидание завершения дочернего процесса. Дело в том, что родительский процесс является лидером сеанса, запущенного на терминале. Преждевременное его закрытие может привести к отлучению дочернего процесса от терминала. т.е. дочерний процесс не сможет считывать и печатать данные на терминал.

А что делать, если родительский процесс захочет принять результат обработки данных дочерним процессом? Попытка закрыть пишущий конец канала и вместо него открыть читающий ничего не даст. Ведь как только закрыли один из дескрипторов, мы сразу теряем к нему доступ. Для того, чтобы организовать обмен данными в обе стороны, нужно открывать два канала: один для передачи данных от родителя к потомку, другой — в обратном направлении.

4. Программные каналы оболочки.

Синтаксис командной оболочки shell позволяет использовать каналы, перенаправляющие стандартный вывод одной программы на стандартный ввод другой.

Например:

bash-3.3$ info pipe | grep PIPE_BUF

Команда info выводит на экран справку по интересующей Вас теме. Например запрос

info pipe

выдаст справку о функции pipe.

Размер этой справки занимает несколько десятков страниц, поэтому, если нас в ней интересует что-то конкретное, например размер канала PIPE_BUF, то найти это будет непросто.

Для поиска нужного места в файле используют другую команду, grep, которая выводит на экран только те строки, которые содержат искомый текст.

Для этого вывод программы info передают на ввод grep. Для передачи используют программный канал, обозначенный вертикальной чертой (|). Так запрос к терминалу

info pipe| grep PIPE_BUF

выведет все строки справочного файла, содержащие слово PIPE_BUF.

Команды, входящие в состав программных каналов, часто называются командами-фильтрами. Команды-фильтры "перехватывают" поток данных, направленный на терминал и преобразовывают его в соответствии со своим алгоритмом, после чего направляют результат на терминал (см. рисунок 6.2).

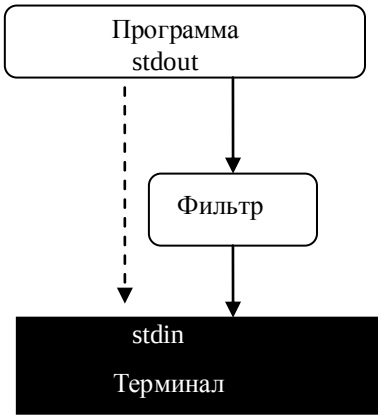


Рисунок 6.2 — Команда-фильтр

Среди команд-фильтров grep самая употребительная. Она применяется везде, где нужно выбрать искомое из большого объема данных. Кроме неё часто употребляются следующие команды:

less - организует вывод информации на экран небольшими порциями. Чтобы прочесть следующую порцию текста, нужно нажать клавишу пробела, а чтобы вернуться к предыдущей порции - клавишу b. Прервать работу программы можно клавишей q.

sort - сортирует строки по алфавиту или порядку номеров.

wc - подсчитывает количество строк, слов, байт или символов в тексте.

tr - заменяет одни символы другими.

cut - вырезает из текста нужные куски и выдает их на стандартный вывод.

head, tail - позволяют ограничить просмотр первыми несколькими строками (head), либо последними несколькими строками (tail).

5. Функции popen и pclose

Эти две функции берут на себя всю рутинную работу, которую мы до сих пор выполняли самостоятельно: создание канала, создание дочернего процесса, закрытие неиспользуемых дескрипторов канала, запуск команды и ожидание завершения команды.

Функция popen() создает канал, затем порождает процесс, исполняющий команду, заданную первым аргументом, и перенаправляет ее стандартный ввод или вывод:

#include <stdio.h>

FILE *popen(const char *command, const char *mode);,

где

command – имя команды, которая выполняет некоторую программу. Между процессом пользователя и вызванным процессом устанавливается канал для обмена информацией.

mode – режим работы канала. Определяется параметром, где "r" означает чтение из канала, а "w" – запись. В случае ошибки, popen() возвращает NULL, иначе – указатель на файл.

Функция pclose() закрывает канал и дожидается окончания связанного с ним дочернего процесса:

#include <stdio.h>

int pclose(FILE *fd);,

где

fd – дескриптор канала, возвращенный функцией popen().

Напишем программу-фильтр, которая принимает в качестве аргумента main() имя команды, считывает данные с её стандартного вывода и преобразует текст к верхнему регистру.

Пример 6.3. Программа-фильтр для команды терминала

```
#include <ctype.h>
#include <stdio.h>
void exit (int);
main(int arc, char *arv[]){
  FILE *fd;
  char ch;
   /* Открываем канал для чтения:                    */
    fd=popen(arv[1],"r");
  /* Читаем данные из канала:                        */
    while(ch=fgetc(fd)!=EOF) {
    printf("%c",toupper(ch));
  }
  pclose(fd);
}
```

Чтобы посмотреть на результат просто откомпилируйте программу и запустите её с именем какой-нибудь команды в качестве аргумента.

Ниже на рисунке 6.3 показан пример такого вывода.

```
bash-3.1$ cc -o filter popen.c
bash-3.1$ filter uname
LINUX
```

Рисунок 6.3. – Результат работы программы-фильтра

Мы можем использовать функцию popen() также и для вывода информации на стандартный вход другой программы. Рассмотрим пример, в котором процесс-отправитель посылает процессу-получателю на стандартный ввод текстовую строку, а процесс-получатель подсчитывает количество символов в ней.

Пример 6.4.

Код процесса-отправителя:

```
#include <stdio.h>
#include <limits.h> // Для PIPE_BUF
main(int arc, char *arv[]){
   FILE *pipe;
   char text[PIPE_BUF];
    printf("Напишите что-нибудь\n");
    fgets(text,PIPE_BUF,stdin);
    pipe=popen("./respond","w");
     fprintf(pipe,text);
     pclose(pipe);
}
```

Код процесса-получателя (respond):

```
#include <string.h>
#include <stdio.h>
#include <limits.h>
main(){
    char str[PIPE_BUF];
    scanf("%s",str);
    int len=strlen(str);
    printf("\n Длина равна %d\n", len);
```

}

Задание:

Модифицируйте пример 6.2 так, чтобы программа запрашивала также пароль у пользователя, сверяла имя и пароль с имеющимися уже в текстовом файле, и в случае несовпадения, предлагала зарегистрироваться, пройдя проверку. В доработанном примере предусмотрите возможность обратной связи дочернего процесса с родительским и согласование их действий по времени.

На основе примеров 6.3 и 6.4 напишите программу-фильтр, выполняющую действия согласно Вашему варианту

Вариант 1. Принимая от команды ls сведения о содержащихся в каталоге файлах, выводит на экран только файлы с расширением .c.

Вариант 2. Принимая от команды ls сведения о содержащихся в каталоге файлах, подсчитывает их количество.

Вариант 3. Принимая от команды ls сведения о содержащихся в каталоге файлах, выводит на экран только те, имена которых содержат указанный фрагмент.

Вариант 4. Принимая от команды info справку по введенной пользователем функции, выводит только первую страницу.

Вариант 5. Принимая от команды info справку по введенной пользователем функции, печатает только те строки, которые содержат название этой функции.

Вариант 6. Принимая от команды info справку по введенной пользователем функции, переписывает её содержимое в указанный файл.

Вариант 7. Выводит только гласные буквы.

Вариант 8. Заменяет во входном потоке одни символы другими. Символы для замены должны быть указаны в качестве аргументов функции main().

Вариант 9. Ищет во входном потоке указанное слово.

Вариант 10. Ограничивает входной поток несколькими последними строками.

Порядок выполнения:

Изучите теоретические сведения.

Изучите примеры программ, приведенных в теоретической части.

Модифицируйте пример 6.2 так, чтобы программа запрашивала также пароль у пользователя, сверяла имя и пароль с имеющимися уже в текстовом файле, и в случае несовпадения, предлагала зарегистрироваться, пройдя проверку. В доработанном примере предусмотрите возможность обратной связи дочернего процесса с родительским и согласование их действий по времени.

На основе примеров 6.3 и 6.4 напишите программу-фильтр, выполняющую действия согласно Вашему варианту (см. ниже).

Протестируйте разработанные Вами программы.

Получите распечатки примеров работы программ.

Оформите отчет.

Форма отчетности:

Отчет по работе содержит:

1. Наименование лабораторной работы;
2. Разработанную программу;
3. Результаты её тестирования;
4. Выводы по работе.

Основная литература

1. Лав, Р. Linux. Системное программирование: учебное пособие / Р. Лав; Пер. с англ. - 2-е изд. - СПб. : Питер, 2014.

Дополнительная литература

2. Молчанов, А. Ю. Системное программное обеспечение : учебник для вузов / А. Ю. Молчанов. - Санкт-Петербург : Питер, 2006.

Контрольные вопросы для самопроверки

1. Для чего служат дескрипторы концов канала?

Какую работу выполняет функция popen()?

Что такое команда-фильтр?

Какой максимальный размер буфера памяти отводится под канал в той системе, в которой Вы работаете ?

Что произойдет, если функции popen() передать имя несуществующей команды?

Лабораторная работа 7.

Файлы FIFO

Цель работы: Изучить способы работы с FIFO файлами. Научиться организовывать взаимодействие процессов посредством FIFO файлов.

Теоретические сведения

1. Общие сведения

FIFO (First In, First Out – первым пришёл, первым ушёл), называемые также именованными каналами, представляют собой особый тип файлов, предназначенный для передачи данных. В отличие от неименованных каналов, доступ к FIFO файлам возможен для любых процессов, том числе и неродственных.

Для каждого FIFO файла на жестком диске создаётся ссылка как на обычный файл. Однако размер его равен нулю, так как блоков для хранения данных у этого файла там нет.

2. Создание и удаление FIFO

Создать FIFO-файл можно как командой с терминала, так и из работающей программы с помощью функции mkfifo():

#include <sys/stat.h>

mkfifo (const *char pathname, int mode);,

где

pathname – имя файла (полное или относительное);

mode – права доступа. Указываются также как и для функции open().

Для создания FIFO с терминала используется одноимённая команда среды исполнения:

mkfifo fifo_name .

3. Чтение и запись в FIFO

После того, как файл FIFO создан, мы можем открыть его функцией open() и далее производить чтение или запись в него с помощью read() и write() или же воспользоваться функциями высокоуровневой библиотеки ввода-вывода.

Ниже в таблице 7.1 приведены функции, которые можно использовать при работе с FIFO.

Таблица 7.1

Два способа работы с FIFO

Функции ввода-вывода	
Низкоуровневые	Высокоуровневые (<stdio.h>)
open() write() read() close()	fopen() fprintf(), fputs(),fputc(), fput() fscanf(), fgets(), gdetc(), fget() fclose()
Дескриптор открытого файла	
целое число: int fd;	структура FILE: FILE * fd

Следующие строки эквивалентны	
fd=open(“myfile”,O_RD ONLY, 0440); close (fd); read (fd,letter,1); write (fd, str, strlen(srt));	fd=fopen(“myfile”,”r”); fclose (fd); fscanf (fd,“%c”,&letter); fprintf (fd,”%s”, str);

Как и в случае с неименованными каналами, если мы попытаемся выполнить запись в FIFO, который не был открыт для чтения, процесс получит сигнал SIGPIPE и завершится по ошибке. Когда последний пишущий в FIFO процесс закроет канал, читающий процесс получит признак конца файла.

Ниже приведены примеры программ, взаимодействующих через FIFO посредством высокоуровневых функций ввода-вывода.

Пример 7.1. Применение высокоуровневых функций для работы с файлами FIFO.

Программа fwriter.c – запись в файл FIFO

```
#include <stdio.h>
#include <unistd.h>
main () {
  FILE*fifo;
  printf(" I am writer \n");
 /* Открываем FIFO:                                  */
  fifo=fopen("./myfifo", "w");
  /* Записываем в FIFO число 211:                    */
  fprintf(fifo, "%d ",211);
  fclose (fifo);
}
```

Программа freader.c – считывание данных из FIFO

```
#include <stdio.h>
main (){
  FILE* fifo;
  int x;
  printf(" I am reader  \n");
  /* Открываем FIFO                                   */
  fifo=fopen("./myfifo", "r");
  /* Считываем из FIFO число x:                       */
  fscanf(fifo, "%d",&buf);
  fclose (fifo);
  printf("result=%d\n",buf);
}
```

Откомпилируем программы:

```
cc –o fwriter  fwriter.c
cc –o freader  freader.c
```

После чего откроем два разных терминала, и в каждом запустим по одной программе. В результате получим следующее(рис. 7.1):

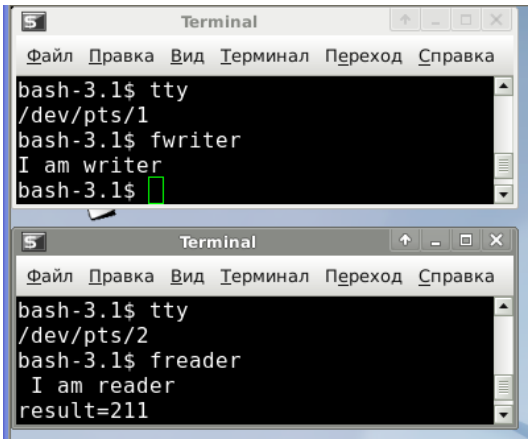


Рис. 7.1. Результат взаимодействия программ freader и fwriter

В первом терминале (dev/pts/1) мы запустили программу-писателя, которая поместила в FIFO число 211, а во втором (dev/pts/2) — программу-читателя, которая извлекла это число и вывела на экран.

Если Вам больше нравиться использовать низкоуровневые функции чтения-записи, то следующий пример для Вас. Для разнообразия заставим программу-писателя принимать 4 текстовые строки от пользователя и переправим их через FIFO программе-читателю.

Пример 7.2. Низкоуровневое взаимодействие с файлами FIFO.

Программа - писатель writer.c:

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#define FILENAME "./myfifo"
main () {
  int fd;
  int i;
  char  message[PIPE_BUF];
  printf("I am writer\n");
  fd=open(FILENAME, O_WRONLY, 0222);
  for (i=0; i<4; i++)
  {
    printf("Write here, please:   \n");
    scanf("%s", message);
    write(fd, message, strlen(message));
    write (1," OK!\n",5);
  }
  close (fd);
}
```

Программа-читатель reader.c:

```
#include <fcntl.h>
#include <malloc.h>
#include <limits.h>
#define FILENAME "./myfifo"
void clear(char *letters, int n);
main (){
  int fd;
  int i=0;
  char* message=(char*) calloc (PIPE_BUF, sizeof(char));
  ssize_t bytes;
  write(1," I am reader\n", 13);
```

```
  for(i=0;i<4;i++)
  {
    fd=open(FILENAME,O_RDONLY, 0640);
    bytes=read (fd, message, PIPE_BUF) ;
    close (fd);
    fprintf (stderr, "put = %s\n", message);
    clear(message, bytes);
   }
}
/************* Очистка строки     ******************/
void clear(char * message, int n)  {
 while (*message)
 {
   * message ='\0';
     message ++;
 }
}
```

Откомпилируйте эти программы также как в предыдущем примере и запустите на разных терминалах. В результате строки, введенные Вами на одном терминале, будет копироваться на другой.

Контрольные вопросы

В чем состоит преимущество именованных каналов перед неименованными?

Что будет в случае прочтения из FIFO меньшего числа байтов, чем находится в канале? Большего числа байтов?

Могут ли 2 и более процессов одновременно читать или записывать в канал?

Какие функции можно использовать для записи в FIFO?

Какие функции можно использовать для чтения из FIFO?

Чему равен размер FIFO на диске?

Порядок выполнения работы

Изучите теоретические сведения.

Ответьте на контрольные вопросы.

Изучите примеры программ, приведенных в теоретической части.

На основе одного из приведенных примеров разработайте программы, моделирующие ситуацию взаимодействия вида клиент-сервер. Один процесс - "сервер" обменивается данными с одним или несколькими "клиентами" посредством FIFO. Все клиенты имеют одинаковый код и могут быть запущены в любом количестве. Способ взаимодействия согласно Вашему варианту указан ниже.

Получите распечатки работы программ.

Оформите отчет.

Отчет должен содержать

– титульный лист;

– формулировку цели работы;

– формулировку задания;

– листинги программ;

– результаты выполнения заданий;

– выводы, согласованные с целью работы;

– ответы на контрольные вопросы.

Варианты заданий

Вариант 1. Молодильная яблоня.

Молодильная яблоня производит случайное число яблок (целое число от 1 до 10) через равные интервалы времени и помещает их в канал. Клиент, сорвавший яблоко, молодеет на год. Каждый клиент на старте определяет свой возраст – случайное число от 100

до 200 лет, затем он потребляет яблоки до тех пор, пока не помолодеет хотя бы на 90 лет. Также клиент пишет в log-файл сообщение вида:

PID… помолодел на … лет и теперь ему … лет.

Предусмотрите задержку по времени на "поедание яблок".

Вариант 2. Кинотеатр.

Написать имитацию службы заказов билетов в кино.

"Кинотеатр" (сервер) формирует таблицу распределения мест в кинозале.

Клиентские процессы посылают в FIFO свой PID. Каждому такому сообщению соответствует один проданный билет. "Кинотеатр" читает канал и уменьшает счетчик выданных билетов. Когда все билеты будут распроданы, он даёт сигнал «отбой» дочерним процессам, дожидается их завершения, закрывает канал, выводит таблицу распределения мест в зале и завершается сам.

Вариант 3. Такси.

Напишите имитацию работы такси. «Диспетчер» помещает в FIFO заявку на поездку. «Такси», которое считывают заявку и отписывается в текстовом файле, сообщая свой номер, после чего имитирует поездку, приостанавливая выполнение на случайный промежуток времени, по окончании поездки, «Такси» пытается считать новую заявку. И так 5 раз, после чего завершается. Диспетчер порождает 20 заявок через равные промежутки времени.

Вариант 4. Служба доставки пиццы.

Напишите имитацию службы доставки пиццы. Пиццерия изготавливает пиццу трех видов по 10 экземпляров каждого. Клиенты пишут в канал заказ на определенный вид пиццы. Когда в службу приходит заказ, она уменьшает счетчик количества экземпляров пиццы соответствующего вида и вычисляет текущую выручку.

Вариант 5. Разведчики.

Написать имитацию службы разведки. Первый процесс (центр) через разные интервалы времени посылает в канал зашифрованное сообщение (введенное пользователем) дочернему процессу. Второй процесс расшифровывает его и пишет одно из 4 сообщений: «все отлично», «пришлите денег», «вас понял», «агент X раскрыт», Затем ситуация повторяется. Обмен сообщениями происходит до тех пор, пока не появится четвертое сообщение. После чего, оба процесса завершаются.

Вариант 6. Интернет-магазин.

Написать упрощенную имитацию работы Интернет-магазина. В магазине имеется каталог товаров: массив a[i] содержит стоимость товаров. При этом значение a[i]=0 означает, что товара в наличии нет. Клиенты через случайные промежутки времени посылают запрос – номер товара, магазин проверяет, есть ли у него этот товар, списывает его, подсчитывает выручку и пишет в канал соответствующее сообщение.

Вариант 7. Сейсмодатчики.

Датчики посылают в FIFO показания колебаний земной поверхности. Показания представляют собой случайные числа от 0 до 10. Центр обработки считывает эти показания. Если более половины датчиков прислали показания от 5 и выше, объявляется чрезвычайная ситуация. Если от четверти до половины – предупреждение, в противном случае обстановка считается спокойной.

Вариант 8. Фрилансеры.

На сайт удалённой работы поступают заказы стоимостью от 100 до 3000 р. Фрилансеры (2-5 человек) принимают заказы, выполняют их и подсчитывают каждый свою прибыль. Написать имитацию взаимодействия фрилансера с сайтом. Процесс, имитирующий работу сайта, генерирует случайные числа через равные промежутки времени и помещает их в FIFO. Процесс-"фрилансер" считывает очередное значение из FIFO, суммирует полученные значения и приостанавливается на случайный промежуток времени. После пятого заказа "фрилансер" печатает в текстовый файл свой PID, выручку и завершает работу.

Вариант 9. Соревнование.

Два класса соревнуются в сборе макулатуры. Ученики приносят стопки бумаг и взвешивают их на весах, после чего вся бумага складывается в общий контейнер. В каждом классе по 20 учеников. Каждый ученик может принести от 1 до 5 кг макулатуры. Написать

имитацию этого соревнования. Процессы-"классы" пишут в FIFO свои идентификаторы и случайные числа из диапазона [1; 5]. Каждой такой записи соответствует одно взвешивание. Процесс-"приемщик" извлекает из FIFO данные и обрабатывает их. После того, как все ученики отметились на весах, объявляется результат.

Вариант 10. Чемодан.

Семья из трех человек собирается на отдых. На троих у них один чемодан. Каждый член семьи кидает в чемодан предметы весом от 0,1 до 4,5 кг. Папа, как самый сильный, может нести вес до 20 кг. Соответственно, как только общий вес чемодана достигнет этого предела, его наполнение прекращается. Написать программную имитацию заполнения чемодана. Три однотипных процесса помещают в FIFO случайные числа от 0,1 до 4,5, попутно подсчитывая количество "своих" предметов, положенных в чемодан. Ещё один процесс извлекает из FIFO числа и вычисляет их сумму. Как только она превысит 20, первым трем процессам даётся сигнал "отбой", после чего процессы выводят результаты и завершаются.

Форма отчетности:

Отчет по работе содержит:

1. Наименование лабораторной работы;
2. Разработанную программу;
3. Результаты её тестирования;
4. Выводы по работе.

Основная литература

1. Лав, Р. Linux. Системное программирование: учебное пособие / Р. Лав; Пер. с англ. - 2-е изд. - СПб. : Питер, 2014.

Дополнительная литература

2. Молчанов, А. Ю. Системное программное обеспечение : учебник для вузов / А. Ю. Молчанов. - Санкт-Петербург : Питер, 2006.

Лабораторная работа 8.

Отображение файла в память

Цель работы: Изучить механизмы отображения файлов в память. Научиться создавать, закрывать и синхронизировать отображения.

Теоретические сведения

1. Использование отображения

Отображение файла в память (memory-mapped files) — это такой способ работы с файлами, при котором содержимое файла записывается в специально подготовленную область памяти.

Отображение файла в память позволяет сократить накладные расходы на чтение/запись, связанные с использованием системных вызовов. При прямой работе с файлами, описанной в лабораторной работе №3 каждая операция чтения/записи требует выполнения достаточно большого количества действий:

чтение данных в системный буфер → модификация данных в буфере → запись в файл.

При отображении же работа состоит только из одного этапа: модификация данных в определённой области памяти.

Отображения файлов часто используются для целей межпроцессного взаимодействия.

Если при создании отображения, объявить его разделяемым, то можно обеспечить доступ к нему одновременно для нескольких процессов.

Дополнительным преимуществом использования отображения является возможность не загружать сразу весь файл, а делать это блоками размером со страницу памяти. Таким образом, даже имея небольшое количество физической памяти, можно легко отобразить файл размером 100 мегабайт или больше.

2. Типы отображений

Существуют 2 типа отображений:

1. Совместно используемое (MAP_SHARED);

2. Закрытое (MAP_PRIVATE).

Совместно используемое отображение, как следует из названия, может быть доступно одновременно нескольким процессам. Изменения, сделанные одним процессом, сразу же становятся видны всем остальным процессам, подключенным к данной области памяти.

Закрытое отображение используется в тех случаях, когда процесс не должен изменять файл на диске. Процесс, подключенный к закрытому отображению, не может в него писать, а только читать. Попытка произвести запись приводит к отключению отображения. Но страницы закрытого отображения содержат все изменения, внесенные другими процессами.

3. Создание отображения

Выделение памяти под отображение и загрузка в неё содержимого осуществляется функцией mmap(), объявленной в заголовочном файле sys/mman.h:

#include < sys/mman.h >

void * mmap(int adress, int size, int protected, int mode, int fd, int seek);,

где

address – адрес в памяти, по которому будет отображаться файл. Если задать пустой адрес NULL, то будет выбран первый подходящий блок памяти.

size – размер выделяемой области памяти (в байтах).

protected – флаги защиты, указывающие на разрешенные операции с данным отображением. Они представляют собой побитовое ИЛИ следующих значений:

PROT_NONE – доступ к этой области запрещен;

PROT_READ – данные можно читать;

PROT_WRITE – в эту область можно записывать;

PROT_EXCL – данные в страницах могут исполняться;

mode – тип отображения. Принимает одно из двух значений:

MAP_SHARED - разделяемое отображение;

MAP_PRIVATE - закрытое отображение.

fd – дескриптор отображаемого открытого файла.

seek – смещение от начала файла. Применяется, если нужно отобразить не весь файл, а только его часть.

Функция mmap() возвращает указатель на область памяти, содержащей отображение.

Для того, чтобы создать отображение, необходимо выполнить следующие действия:

1) Открыть файл функцией open() (см ЛР №3).

2) Объявить указатель на будущее отображение. Тип указателя зависит от типа данных, хранящихся в файле.

3) Открыть отображение.

4) После того, как отображение открыто, держать файл в памяти уже не нужно, поэтому его обычно закрывают.

Пример 8.1. Создание разделяемого отображения текстового файла "test.txt":

```
#include <sys/mman.h>
…
int fd;
fd=open("file.txt", O_RDWR,0600);
char *buf;   //Указатель на отображение
buf=mmap(NULL,4096, PROT_READ, MAP_SHARED,fd,0);
if  (buf== MAP_FAILED)  {
    printf(" Не удалось создать отображение \n");
  return ;
}
close(fd);
…
```

/* Далее следуют операции с отображением */

Доступ к элементам отображения осуществляется так же, как к элементам массива: обращением к buf[i].

4. Сохранение внесенных изменений

Работая с отображением, мы можем вносить в него правки. При этом содержание исходного файла на диске останется прежним, ведь мы воздействовали лишь на образ в оперативной памяти. Для того чтобы сохранить внесённые изменения, нужно скопировать образ в файл.

Процесс приведения содержимого файла в соответствие с его же отображением называется синхронизацией изменений.

Чтобы выполнить синхронизацию, вызывают функцию msync():

#include < sys/mman.h >

int msync(void *start, size_t length, int flags);,

где

start – адрес в памяти, содержащей отображение;

length – длина синхронизируемого участка памяти, измеряется в байтах;

flags – параметр, управляющий ходом синхронизации.

Флаги представляют собой битовую комбинацию MS_INVALIDATE и MS_SYNC или MS_ASYNC.

Последние два параметра никогда не применяются одновременно.

MS_SYNC даёт системе задание записать изменения и ждёт его исполнения;

MS_ASYNC даёт системе задание записать изменения и немедленно возвращается в ожидающий его процесс;

MS_INVALIDATE приказывает обновить системе другие отражения этого же файла так, чтобы они содержали изменения, внесенные этим вызовом.

Как можно видеть из описания функции msync(), она может обновлять не весь файл, а только выбранную его часть.

Примечания.

1. Для того, чтобы синхронизация была возможной, используйте флаг MAP_SHARED в функции mmup().

2. Вызов функции msync() не приводит к немедленной синхронизации. Она лишь даёт нам гарантию, что это произойдёт до закрытия отображения.

Пример 8.2. Синхронизация первых 10 байт отображения buf:

msync(buf,10,MS_SYNC|MS_INVALIDATE);

5. Закрытие отображения

После окончания работы с файлом процесс может закрыть его отображение с помощью munmap(). Это приводит к тому, что последующие доступы к этому адресу будут генерировать ошибку сегментирования (SIGSEGV) и высвобождает память. Закрыть можно как всё отражение, так и его часть, начиная с некоторого адреса в памяти:

#include < sys/mman.h >

int munmap(void *start, size_t length);,

где

start – адрес начала области памяти для отмены отображения;

length – размер высвобождаемой части памяти.

6. Этапы работы с отображением

Процесс открывает файл и получает его дескриптор.

Создаётся отображение. Его адрес записывается в специально объявленную для этого переменную - указатель. Тип указателя соответствует типу обрабатываемых данных.

В каждом процессе отображение выглядит как массив. Чтение/запись элемента отображения происходит при обращении к соответствующему элементу массива. Например, если в программе под отображение отведён массив buf, то выражение buf[0] инициирует чтение первого элемента, а buf[i]=’q’– запись в i-й элемент значения ‘q’.

Чтобы сохранить произведённые с отображением изменения, процесс вызывает функцию msync().

По окончании работы с отображением его закрывают.

Для примера, приведём код программы, открывающей отображение текстового файла, печатающий содержимое файла и изменяющей его первый символ. Имя отображаемого файла оформим как аргумент argv[1] программы. Чтобы не загромождать код, из него исключена часть проверок.

Пример 8.3. Работа с отображением файла

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
/**** Вывод содержимого отображения на экран  *****/
void show_map(char*buf) {
   while(*buf) {
       printf ("%c",*buf);
       buf++;
   }
}
main() {
int fd,  size=4096;
char *buf; //Указатель на отображение текстового файла
/* Открываем файл для чтения и записи:                    */
fd=open("file.txt", O_RDWR|O_CREAT,0600);
/* Создаём отображение:                                    */
buf=mmap(NULL, size, PROT_READ|PROT_WRITE,
  MAP_SHARED,fd,0);
if  (buf== MAP_FAILED){
     printf("Не удалось создать отображение \n");
     return ;}
/* Теперь файл можно закрывать:                            */
close (fd);
/* Выводим содержимое отображения на экран :          */
show_map(buf);
/* Заменяем первый символ на тот,
    что введёт пользователь:                               */
printf("Выберите букву: \n");
buf[0]=fgetc(stdin);
/* Синхронизируем изменения, произведённые
   с первым байтом отображения:                            */
msync(buf,1,MS_SYNC|MS_INVALIDATE);
show_map(buf);
/* Закрываем отображение:                                  */
munmap(buf,size);
}
```

Если открыть два разных терминала и запустить на каждом по экземпляру этой программы, то можно увидеть, как изменения, сделанные одним экземпляром в тексте файла, становятся видны другому.

Контрольные вопросы

В чем состоит преимущество использования отображения файла перед прямым доступом к нему?

В каких случаях используется разделяемое отображение?

Что будет, если в приведённом примере указать тип отображения как MAP_PRIVATE?

Что такое синхронизация отображения?
Какую работу выполняет функция mmap()?
Для чего используется вызов функции munmap()?

Порядок выполнения работы
Изучите теоретические сведения.
Ответьте на контрольные вопросы.
Изучите пример программы, приведенный в теоретической части.
На основе изученного примера напишите программы для работы с отображением файла согласно Вашему варианту.
Получите распечатки работы программ.
Оформите отчет.
Отчет должен содержать
– титульный лист;
– формулировку цели работы;
– формулировку задания;
– листинги программ;
– результаты выполнения заданий;
– выводы, согласованные с целью работы;
– ответы на контрольные вопросы.
Варианты заданий
Вариант 1.
Первая программа заменяет все символы, стоящие на чётных местах пробелами, вторая – переводит все нечётные символы в верхний регистр. Обе программы выводят на экран промежуточный результат.
Вариант 2.
Напишите две программы, работающие с отображением текстового файла в память.
Первая программа заменяет первые 20 букв цифрами, а вторая – дописывает в конец файла слово, введённое пользователем.
Вариант 3.
Первая программа пишет в отображение числа, введенные пользователем. Вторая по запросу пользователя находит их сумму.
Вариант 4.
Первая программа пишет в файл квадраты чисел от 1 до 20. Вторая по запросу пользователя выводит на терминал число с заданным порядковым номером.
Вариант 5.
Первая программа каждые 5 секунд выводит содержимое отображения файла на экран, вторая позволяет пользователю редактировать содержимое файла с терминала.
Вариант 6.
Первая программа выводит содержимое отображения файла на экран, вторая сортирует его по возрастанию.
Вариант 7.
Первая программа выводит содержимое отображения файла на экран, вторая записывает его "задом наперёд".
Вариант 8.
Первая программа подсчитывает количество символов, указанных пользователем. Вторая добавляет в конец файла слово, указанное пользователем.
Вариант 9.
Первая программа выводит содержимое отображения файла на экран, вторая организует циклическое смещение символов в указанном фрагменте файла.
Вариант 10.
Первая программа каждые 5 секунд выводит содержимое отображения файла на экран, вторая - заменяет букву с номером i на другую. Номер и букву вводит пользователь.

Форма отчетности:
Отчет по работе содержит:
1. Наименование лабораторной работы;
2. Разработанную программу;
3. Результаты её тестирования;
4. Выводы по работе.

Основная литература

1. Лав, Р. Linux. Системное программирование: учебное пособие / Р. Лав; Пер. с англ. - 2-е изд. - СПб. : Питер, 2014.

Дополнительная литература

2. Молчанов, А. Ю. Системное программное обеспечение : учебник для вузов / А. Ю. Молчанов. - Санкт-Петербург : Питер, 2006.

Лабораторная работа 9.

Разделяемая память

Цель работы: Изучить механизмы работы разделяемой памяти, освоить способы её применения для межпроцессного взаимодействия.

Теоретические сведения

1. Общие сведения

Разделяемая, общая или совместно используемая память (shared memory) является наиболее эффективным механизмом межпроцессного взаимодействия.

Каждому процессу в системе выделяется свое адресное пространство, в пределах которого он и может функционировать. Попытка доступа к памяти вне этого адресного пространства приводит к появлению сигнала SIGSEGV (ошибка сегментации) с завершением процесса. Поэтому для того, чтобы передавать данные между процессами, необходимо задействовать системные вызовы и буферы ядра.

Разделяемая память позволяет двум и более процессам свободно пользоваться одним и тем же кусочком адресного пространства. Она представляет собой зарезервированный сегмент в виртуальной памяти, доступный набору процессов. Изменения, записанные в разделяемую память одним процессом, сразу же становятся очевидными для всех других процессов.

С каждым разделяемым сегментом памяти связаны два числа:

ключ – идентификатор сегмента в системе. Процессы могут присоединяться к имеющемуся сегменту памяти, если знают его ключ. Процесс может создать сегмент для личного пользования (им самим и его потомками), указав ключ IPC_PRIVATE;

идентификатор сегмента – служит для обращения к сегменту внутри процесса.

Порядок работы процесса с разделяемой памятью следующий:

1) Сначала процесс создаёт общий сегмент памяти или же получает доступ к уже существующему. Доступ осуществляется через идентификатор, возвращаемый функцией shmget().

2) После получения идентификатора к общей памяти надо подключиться. Эта операция называется присоединением сегмента. Она возвращает его адрес в виртуальном адресном пространстве процесса. Виртуальный адрес одного и того же сегмента может быть разным для разных процессов.

3) Далее процессы ведут чтение и запись данных в сегменте разделяемой памяти, используя для этого указатели на него.

4) Процесс, закончивший работу с сегментом разделяемой памяти должен "отсоединить" его. По окончании использования сегмента всеми процессами сегмент должен быть уничтожен.

2. Создание сегмента разделяемой памяти и получение его идентификатора.

Для получения идентификатора сегмента общей памяти используется функция shmget() (англ. shared memory get – получить общую память). Она же создаёт новый сегмент, проверяет наличие уже существующего и определяет права доступа к нему, если того захочет программист. Формат функции следующий:

#include <sys/shm.h>

int shmget (key_t key, int size, int flag);,

где

key – ключ сегмента. Можно закрыть доступ к сегменту со стороны внешних процессов, если задать в качестве ключа значение IPC_PRIVATE.

size – размер памяти, выделяемой под сегмент. Фактически выделяемый размер памяти округляется до в большую сторону до целого числа страниц (т.е. числа, кратного _SC_PAGESIZE);

flag – поведение функции shmget().

Параметр flag может содержать следующие значения:

IPC_CREAT – служит для создания нового сегмента. Если он не задан, то функция shmget() будет искать сегмент, соответствующий ключу key; затем она проверит, имеет ли пользователь права на получение идентификатора, соответствующего этому сегменту, и удостоверится, не помечен ли сегмент на удаление.

IPC_EXCL – проверяет наличие общего сегмента, использующего данный ключ. Если сегмент уже существует, то будет сгенерирована ошибка. Обычно используется совместно с IPC_CREAT для предотвращения повторной попытки создать уже существующий сегмент.

mode – режим доступа, аналогичный режиму доступа к файлу. По-умолчанию устанавливается в 0600.

Если требуется получить доступ к уже существующему сегменту, то флаги IPC_CREAT и IPC_EXECL не устанавливаются.

Пример 9. 1. Создание одной страницы общей памяти с ключом равным 8 и получение её идентификатора:

id= shmget(8, _SC_PAGESIZE, IPC_CREAT|IPC_EXCL|0640);

Пример 9. 2. Получение идентификатора существующей страницы памяти с ключом 8:

id= shmget(8, _SC_PAGESIZE, 0640);

Рассмотрим отдельно вопросы, связанные с получением ключа к сегменту.

Ключ – это целое положительное число, выбираемое программистом произвольным образом. Однако всегда есть вероятность, что данный ключ уже используется в системе другими процессами и для других целей. В этом случае при попытке создать сегмент программа потерпи неудачу, а при подключении к сегменту есть риск повредить данные.

Один из способов формирования ключа является использование функции ftok(). Она позволяет привязать создаваемый ключ к имени существующего файла:

#include <sys/ipc.h>

key_t ftok(const char* pathname, int proj_id);

Здесь

pathname – имя файла. Файл должен существовать и быть доступным со стороны процесса.

proj_id – идентификатор проекта.

Константа, идентифицирующая группу процессов, которые могут получить доступ к общему сегменту памяти. Выбирается программистом произвольным образом:

key=ftok("file.txt", 'A');

Функция ftok возвращает одинаковые ключи для разных ссылок на один и тот же файл, если задается одно и то же значение proj_id, и разные ключи, если задаются одинаковые имена файлов, но разные значения proj_id. Однако нет никакой гарантии, что для разных файлов будут возвращены разные ключи.

Пример 9. 3. Выделение 4096 байт общей памяти с ключом, привязанным к файлу file.txt:

id=shmget(ftok("file.tx",'A'), 4096, IPC_CREAT|IPC_EXCL);

Еще один способ назначить ключ сегменту, это указать в качестве первого параметра функции shmget() значение IPC_PRIVATE. Использование значения IPC_PRIVATE всегда приводит к попытке создания нового сегмента разделяемой памяти с заданными правами доступа и с ключом, который не совпадает со значением ключа ни одного из уже существующих сегментов и который не может быть получен с помощью функции ftok() ни при одной комбинации ее параметров. Наличие флагов IPC_CREAT и IPC_EXCL в этом случае игнорируется. Доступ к сегменту. созданному с применением IPC_PRIVATE будет возможен лишь для самого процесса и его потомков.

3. Присоединение общего сегмента

Получив идентификатор разделяемого сегмента памяти, процесс должен "присоединить" сегмент к своему виртуальному адресному пространству. Эта операция выполняется функцией shmat()(shared memory attach):

#include <sys/shm.h>

void *shmat(int id, const void *addr, int flag);

где

id – идентификатор сегмента, полученный ранее функцией shmget().

addr – адрес, по которому присоединяется сегмент.

Если addr равен NULL, то система выбирает для сегмента подходящий (неиспользованный) адрес.

flag – задает параметры подключения:

SHM_RND указывает, что адрес, определенный для второго параметра, должен быть округлен назад к множителю размера страницы. Если Вы не указываете этот флаг, Вы должны выровнять на границу страницы второй параметр передаваемый shmat самостоятельно.

SHM_RDONLY указывает, что сегмент будет доступен только для чтения. Если этот флаг не задан, то для сегмента будут доступны и чтение и запись. Сегментов "только-запись" не существует.

Биты прав доступа. Они задаются также как и для файлов.

4. Отсоединение сегмента

По окончании работы с разделяемой памятью, её нужно отсоединить с помощью функции shmdt()(shared memory detach):

#include <sys/shm.h>

int shmdt(const void *adrr);

Функция shmdt() отстыковывает сегмент разделяемой памяти, находящийся по адресу addr, от адресного пространства вызывающего процесса.

5. Прочие операции над сегментом общей памяти

Все прочие операции выполняет системный вызов shmctl().

Он предназначен для получения информации об области разделяемой памяти, изменения её атрибутов и удаления её из системы:

#include <sys/shm.h>

int shmctl(int id, int cmds, struct shmid_ds *buf);,

где

id – идентификатор сегмента;

cmds – выполняемая операция. Может принимать значения :

IPC_RMID – удалить сегмент;

IPC_STAT – получить информацию о сегменте;

IPC_SET – изменить атрибуты сегмента.

buf — указатель на структуру shmid_ds, содержащую информацию о сегменте. Если требуется просто удалить сегмент, то можно использовать пустой указатель NULL.

Пример 9.4. Удаление сегмента с идентификатором id shmctl(id, IPC_RMID,NULL);

В качестве иллюстрации к сказанному, рассмотрим две программы, обращающиеся к одному и тому же участку памяти.

Первая программа создает сегмент и пишет в него целые числа от 10 до 1:

Пример 9.1. Программа shm1.c:

```
# include <stdio.h>
# include <unistd.h>
#include<sys/ipc.h>
# include <sys/shm.h>
int main()    {
int id,  size, i;
int *buf;     // тип buf зависит от разделяемых данных
/* Ключ сегмента положим равным 99                          */
key_t key=99;
/* Создаем общий сегмент памяти                              */
id=shmget (key, _SC_PAGESIZE, IPC_CREAT| IPC_EXCL|
0770  );
/* Активируем сегмент и связываем его с массивом buf  */
buf = (int *) shmat(id, NULL, 0);
/*  Запись и чтение сегмента                                  */
for (i=10; i>0; i--) {
fprintf(stderr, "New:..%d\n", *buf); // чтение
*buf=i;                                  //запись
sleep(1);
}
printf ("The end \n");
/* Отсоединяем сегмент                                       */
shmdt (buf);
/* Помечает сегмент на удаление                              */
shmctl (id, IPC_RMID, NULL);
return 0;
}
```

Наша программа создаёт сегмент памяти размером в одну страницу (_SC_PAGESIZE, строка 11) и присваивает ей права доступа 0770. В строке 24 этот сегмент уничтожается. Если этого не сделать, то при повторном запуске программы произойдёт сбой, так как программа будет пытаться создать сегмент, который уже существует в памяти.

В цикле for (строки 15-19) производится чтение текущего значения, содержащегося в сегменте и запись в него новых значений. Эта программа, как мы можем видеть, производит обратный отсчет от 10 до 1.

Если скомпилировать и запустить эту программу, то она выведет в терминале следующее:

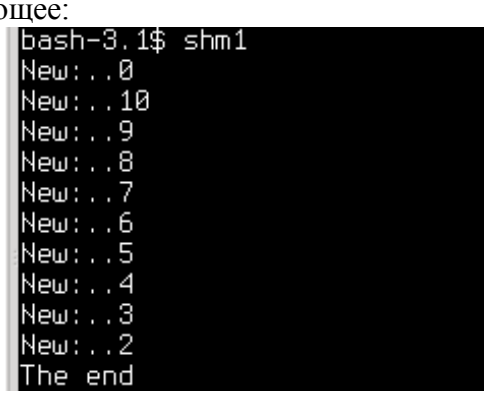


Рисунок 9.1– Результат работы программы shm1

Вторая программа, подключаясь к сегменту, выхватывает из него текущее значение, помещенное первой программой, печатает его и завершается.

Пример 9.2. Программа shm2.c:

```
#include <stdio.h>
#include <sys/shm.h>
```

```
int main()
{
int id;
int *buf;
// Получаем доступ к сегменту с ключом 99:
id = shmget(99, _SC_PAGESIZE, 0777);
//Присоединяем сегмент
buf = (int *)shmat(id, NULL, 0);
// Считываем значение:
printf("Information:..%d", *buf);
*buf=0;
// Отсоединяем сегмент:
shmdt(buf);
return 0;
}
```

Если запустить эту программу в отдельном терминале параллельно с первой, то она выхватит из общего сегмента текущее значение (рис.9.2).

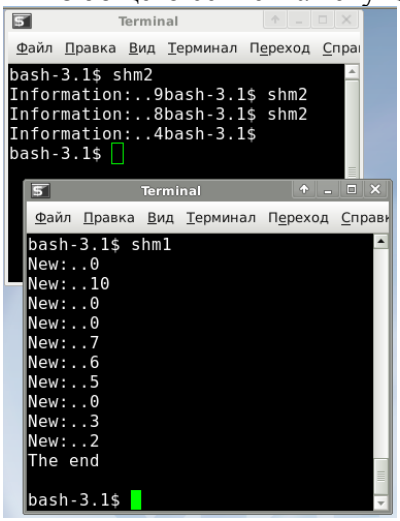


Рис. 9.2. Результат параллельного запуска программ shm1 и shm2

Обратите внимание на то , что вторая программа не пытается создать сегмент, а обращается к уже существующему. Поэтому при тестировании, Вы должны сначала запустить программу shm1, и только потом shm2.

6. Отладка программ

Сегменты разделяемой памяти существуют отдельно от процесса и, если их явно не уничтожить, могут накапливаться в памяти в течение многих дней. Если общее число таких областей или их суммарная память превысит критическое значение, попытка создать новый сегмент приведёт к отказу.

Если Вы хотите посмотреть список всех существующих на текущий момент сегментов, наберите в терминале команду

ipcs -m.

В результате выполнения этой команды на экран будет выведена информация о сегментах общей памяти как на рисунке 9.3:

```
bash-3.1$ ipcs -m

------ Сегменты совм. исп. памяти --------
ключ    shmid      владелец права байты nattch     состояние
0x00000000 0          katty     600        393216     2
0x00000000 32769      katty     600        393216     2
0x00000000 65538      katty     600        393216     2
0x00000000 98307      katty     600        393216     2
0x00000000 131076     katty     600        393216     2
0x00000000 163845     katty     600        393216     2
0x00000000 196614     katty     600        393216     2
```

Рис.9.3. Таблица используемых сегментов разделяемой памяти

Вторая колонка этой таблицы (shmid) содержит идентификатор сегмента, а в шестой указано количество процессов, в данный момент использующих сегмент. На рисунке 9.3 все сегменты имеют по два подключенных к ним процесса. Чтобы сегменты не накапливались, их нужно своевременно удалять. Для этого используйте команду терминала

ipcrm -m shmid,

где shmid – идентификатор сегмента.

Контрольные вопросы

Что такое разделяемая память?

Для чего используется ключ (key) сегмента?

Каким атрибутами обладает общий сегмент памяти?

Какая функция создаёт сегмент разделяемой памяти?

Для каких целей используют функцию shmat()?

Что может произойти, если программист забудет удалить разделяемый сегмент памяти по окончанию работы?

Как узнать, сколько разделяемых сегментов использует в данный момент система?

Порядок выполнения работы

Изучить теоретические сведения.

Изучить примеры программ, приведенных в теоретической части.

Модифицировать примеры 9.1 и 9.2 для выполнения задания согласно варианту.

Посмотреть через Терминал список общих сегментов памяти. Если среди них появился сегмент, созданный Вашими программами, удалить его.

Оформить отчет.

Отчет должен включать:

– титульный лист;

– формулировку цели работы;

– формулировку задания;

– листинги программ;

– результаты выполнения программ;

– выводы, согласованные с целью работы;

– ответы на контрольные вопросы.

Варианты заданий

Вариант 1.

Напишите две программы, взаимодействующие через разделяемую память. Первая программа помещает в разделяемую память 10 случайных чисел, вторая считывает их и находит наибольшее.

Вариант 2.

Напишите две программы, взаимодействующие через разделяемую память. Первая программа помещает в разделяемую память содержимое текстового файла, вторая — ищет в нём указанный символ.

Вариант 3.

Напишите две программы, взаимодействующие через разделяемую память. Первая программа пишет в разделяемую память 20 случайных чисел, вторая считывает их, находит их сумму и записывает результат в файл.

Вариант 4.

Напишите две программы, взаимодействующие через разделяемую память. Первая программа помещает в разделяемую память содержимое текстового файла, вторая — складывает коды символов по модулю 26 и печатает результат.

Вариант 5.

Напишите две программы, взаимодействующие через разделяемую память. Обе программы пишут в разделяемую память по 10 чисел: первая чётные, вторая нечётные. После чего первая программа выводит на экран содержимое общего сектора памяти.

Вариант 6

Напишите две программы, взаимодействующие через разделяемую память. Обе программы считывают текстовые файлы, каждая свой, и построчно записывают их содержимое в разделяемую память. После чего первая программа выводит на экран содержимое общего сегмента памяти.

Вариант 7

Напишите программу, создающую разделяемый сегмент с ключом IPC_PRIVATE и порождающую дочерний процесс. Родительский процесс пишет в разделяемую память содержимое некоторого файла, дочерний – считывает его и выводит на экран.

Вариант 8

Напишите программу, создающую разделяемый сегмент с ключом IPC_PRIVATE и порождающую дочерний процесс. Родительский процесс помещает в общий сегмент случайные числа, дочерний – печатает их и находит наименьшее.

Вариант 9

Напишите программу, создающую разделяемый сегмент с ключом IPC_PRIVATE и порождающую дочерний процесс. Родительский процесс пишет в разделяемый сегмент 10 нечётных чисел, дочерний – 10 чётных. После завершения дочернего процесса, содержимое общего сегмента выводится на экран.

Вариант 10

Напишите программу, создающую разделяемый сегмент с ключом IPC_PRIVATE и порождающую дочерний процесс. Родительский процесс пишет в разделяемый сегмент 10 случайных чисел, дочерний – уменьшает все четные числа на

Форма отчетности:

Отчет по работе содержит:

1. Наименование лабораторной работы;
2. Разработанную программу;
3. Результаты её тестирования;
4. Выводы по работе.

Основная литература

1. Лав, Р. Linux. Системное программирование: учебное пособие / Р. Лав; Пер. с англ. - 2-е изд. - СПб. : Питер, 2014.

Дополнительная литература

2. Молчанов, А. Ю. Системное программное обеспечение : учебник для вузов / А. Ю. Молчанов. - Санкт-Петербург : Питер, 2006.

9.2. Методические указания по выполнению курсовой работы

Курсовая работа по дисциплине Языки и методы программирования представляет письменный отчет по самостоятельной работе обучающегося в рамках заданной темы.

Основными целями курсовой работы являются:

- закрепление практических умений в области разработки программного обеспечения;
- углубление теоретических и практических знаний по дисциплине;
- развитие способности находить нужную информацию и применять её для решения - поставленных задач;
- формирование навыков планирования, реализации, анализа и оценки собственной деятельности.

Задачи курсовой работы определяются исходя из индивидуального задания, полученного обучающимся. В общем случае, задачи курсовой следующие:

- изучить литературу по выбранной теме;
- проанализировать имеющиеся на данный момент алгоритмы, методы средства решения поставленной задачи;
- разработать программное приложение, полученное в виде задания на разработку.

- при необходимости, разработать собственные методы и алгоритмы решения поставленной задачи.

Этапы работы над курсовой

1. Выбор технического задания на разработку и формулировка соответствующей ему темы;
2. Подбор литературных источников по теме.
3. Составление предварительного варианта плана.
4. Написание текста курсовой работы и разработка приложений.
5. Оформление курсовой работы
6. Составление доклада и разработка презентации .
7. Защита курсовой работы.

Требования к качеству текста

Текст курсовой работы должен быть логичным и последовательным. Расплывчатые и объемные рассуждения авторов произведений, используемых в работе, следует представить лаконично и обоснованно. Каждое предложение должно быть литературно обработано. Стиль изложения материала должен быть единым по всей работе. Орфографические, грамматические и стилистические ошибки недопустимы.

Не рекомендуется вести изложение от первого лица: «Я считаю», «Мне кажется», «Я получил(а)». В отдельных случаях возможно использование выражений типа: «По нашему мнению», «По мнению автора выпускной квалификационной работы». Однако более предпочтительными оказываются фразы «на основе выполненного анализа можно утверждать», «проведенные исследования подтвердили…».

Сокращения в виде аббревиатуры допускаются только при условии их первоначального разъяснения.

Заголовки глав и параграфов должны быть короткими и содержательными, они не должны повторять названия темы курсовой работы.

Содержание глав основной части должно точно соответствовать теме работы и полностью её раскрывать.

В целом текст работы должен продемонстрировать умение автора сжато, точно, логично и аргументировано излагать информацию.

Требования к содержанию работы

Титульный лист оформляется по единому образцу, принятому на выпускающей кафедре. Образец оформления титульного листа представлен ниже:

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ

«БРАТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Кафедра математики

КУРСОВАЯ РАБОТА

по дисциплине
Системное программирование

Тема:

Выполнил:
студент гр._____ _____ _____
группа *подпись* *Ф.И.О. студента*

Проверил:
ст. преподаватель каф. математики _____
Ф.И.О. преподавателя

Братск 2017

Содержание включает перечисление частей работы, начиная от введения и заканчивая приложениями, с указанием страницы начала каждой части. Названия частей курсовой должны в точности соответствовать содержанию

Введение

Во введении, объем которого 1- 1,5 страницы печатного текста, указывается актуальность темы, формулируются цели и задачи работы. Здесь же приводится аннотация глав работы.

Первая глава

Содержанием первой главы являются, как правило, основные теоретические вопросы по выбранной теме.

Возможные структурные элементы главы:

1. анализ рассматриваемой проблемы (математической или информационной);
2. место и роль данной темы в науке (математике, информатике);
3. обзор алгоритмов и методов, применяемых для решения задач, схожих с поставленной;
4. получение и представление собственных результатов (теоретических и практических);
5. предположения и выводы.

Вторая глава

Содержит

постановку задачи на разработку программы,

1. подходы к решению проблемы: алгоритмы, методы, средства разработки;
2. описание функционала программной разработки, её пользовательского интерфейса,

3. общую структуру проекта;
4. набор тестовых примеров и запланированные результаты их применения;
5. описание результатов запуска и тестирования программы со скриншотами.

Код вставляется фрагментами с обязательным текстовым комментарием.

Заключение - последовательное логически стройное изложение итогов и их соотношение с общей целью и конкретными задачами, поставленными и сформулированными во введении. В заключении концентрированно отражаются все выводы и положения, содержащиеся в работе. Объём заключения от 2/3 страницы.

Список литературы

Список использованной литературы должен быть выполнен в соответствии с ГОСТ 7.32.2001 «Система стандартов по информации, библиотечному и издательскому делу. Отчет о научно-исследовательской работе. Структура и правила оформления» и правилами библиографического описания документов ГОСТ 7.1-2003 «Библиографическая запись. Библиографическое описание». ГОСТы имеются в библиотеке университета, где можно получить и консультацию по оформлению списков литературы. Кроме того, подробное оформление библиографического списка имеется на сайте библиотеки БрГУ.

Список обязательно должен быть пронумерован. Каждый источник упоминается в списке один раз, вне зависимости от того, как часто на него делается ссылка в тексте работы.

Наиболее удобным является алфавитное расположение материала, так как в этом случае произведения собираются в авторских комплексах. Произведения одного автора расставляются в списке по алфавиту заглавий.

Официальные документы ставятся в начале списка в определенном порядке: Конституции; Кодексы; Законы; Указы Президента; Постановление Правительства; другие нормативные акты (письма, приказы и т.д.). Внутри каждой группы документы располагаются в хронологическом порядке.

Литература на иностранных языках ставится в конце списка после литературы на русском языке, образуя дополнительный алфавитный ряд.

Для каждого документа предусмотрены следующие элементы библиографической характеристики: фамилия автора, инициалы; название; подзаголовочные сведения (учебник, учебное пособие, словарь и т. д.); выходные сведения (место издания, издательство, год издания); количественная характеристика (общее количество страниц в книге) (Приложение 5).

Требования к оформлению курсовой работы

1) текст работы печатается на одной стороне листа белой бумаги *формата А4;*

2) *объем* – 25–30 страниц.

3) *поля:* левое – 3 см, правое – 1,0 см, верхнее - 2,0 см, нижнее – 2,0 см;

4) *шрифт* Times New Roman по всей работе;

4) *размер букв*: основной текст –14, в таблицах и приложениях допускается использовать кегль 12;

5) *цвет* шрифта – черный;

6) *интервал* между строками – 1,5, в таблицах допустим одинарный интервал;

7) в*ыравнивание* по ширине;

8) *автоматические переносы* обязательны;

9) абзацный отступ – 1,27 см.;

10) все страницы работы нумеруются, начиная с титульного листа и до последней страницы приложений, номер страницы на титульном листе не проставляют;

11) разрешается использовать компьютерные возможности акцентирования внимания на определенных частях текста, терминах и определениях, применяя курсив.

Введение, заключение, литература, названия глав, параграфов и приложений печатаются **жирным** шрифтом по *левому краю листа с абзацным отступом 1,27см*.

Слова «Содержание», «Введение», «Заключение», «Литература», названия глав и параграфов печатаются с прописной буквы, последующие буквы – строчные.

После названий глав и параграфов *точки не ставятся*. Не рекомендуется подчеркивать заголовки.

В заголовках *не допускается* перенос слов, сокращение слов и применение аббревиатур. Если заголовок состоит из двух предложений, их разделяют точкой, а после второго предложения точка не ставится.

Не допускается расположение заголовка на одной странице, а текста – на другой. Если заголовок размещается в нижней части страницы, то после него должно быть не менее трех строк текста. В противном случае, заголовок и текст переносятся на следующую страницу.

Главы и параграфы нумеруются арабскими цифрами. Принадлежность параграфа главе отмечается двумя цифрами, первая из которых – номер главы, а вторая (через точку) – номер параграфа. Например, «1.2 Информационные технологии в экономике» – второй параграф первой главы. Нумерация параграфов едина внутри главы. В каждой главе должно быть не менее двух параграфов.

Если имеется необходимость выделить части параграфа (не менее двух), их тоже нумеруют. К примеру, заголовок «1.2.1 Система 1С:бухгалтерия» представляет первую часть второго параграфа первой главы. И таких частей должно быть не менее двух.

Между заголовком главы и параграфа, между заголовком параграфа и последующим текстом – полуторный интервал. Каждая глава, введение, заключение, список литературы и приложения начинаются с новой страницы.

Рисунки, графики, схемы, таблицы могут иметь либо сквозную нумерацию по всей работе, либо нумерацию, как и параграфы, соответствующую данной главе.

При наличии в тексте списков перед каждой позицией следует ставить дефис (–), другие маркеры не допускаются, запись производится с абзацного отступа.

Если на перечисления в тексте делаются ссылки, то необходимо использовать строчную букву или арабские цифры, после которых ставится скобка (если пункты перечисления будут разделены точкой с запятой; в этом случае каждый пункт начинается со строчной буквы).

Если каждый пункт содержит в себе законченную мысль (одно или несколько предложений) в этом случае каждый пункт начинается с прописной буквы и в конце ставится точка. Пункты списка не должны содержать несколько предложений.

Иллюстрации, за исключением иллюстраций приложений, следует нумеровать арабскими цифрами без символа «№». Если рисунок один, то он обозначается словом «Рисунок». Например, «Рисунок 1 – Окно сохранения документа». Слово «Рисунок», его номер и наименование располагают по центру строки под изображением без точки в конце.

Нумерация рисунков может быть как сквозная, так и в пределах главы. В последнем случае номер иллюстрации состоит из номера главы и порядкового номера иллюстрации, разделенных точкой, например, «Рисунок 1.1».

Иллюстрации (чертежи, графики, схемы, компьютерные распечатки, диаграммы, фотоснимки) следует располагать в ВКР непосредственно после текста, в котором они упоминаются впервые, или на следующей странице.

Ссылка на рисунок должна предшествовать рисунку. При ссылках на иллюстрации следует писать «… в соответствии с рис. 2» при сквозной нумерации и «…в соответствии с рис. 1.2» при нумерации в пределах главы. Ссылка на рисунок может заключаться в круглые скобки: (рис. 2) или (рис. 1.2), соответственно. Схемы, диаграммы и графики тоже называются рисунками.

Разрешается выполнять иллюстрации в любых цветах на цветном принтере, обеспечивающем хорошее качество печати. Кроме формата А4 для иллюстраций (включая таблицы) разрешается использовать бумагу большего формата вплоть до А3. Такой лист складывается соответствующим образом до формата А4 и рассматривается как приложение. При нумерации он учитывается как одна страница.

Название таблицы следует помещать над таблицей с выравниванием по левому краю с абзацным отступом в одну строку с ее номером и названием через тире, например, «Таблица 2 – Сравнение показателей». Если название таблицы не помещается в одну строку, то на второй строке название начинается под заглавной буквой первой строки. На все таблицы

должны быть ссылки в тексте ВКР. При ссылке следует писать слово «таблица» с указанием ее номера, например, «…показано в таблице 2» или (таблица 1.2).

Таблицу с большим количеством строк допускается переносить на другую страницу. При переносе части таблицы на другую страницу слово «Таблица» и ее номер указывают один раз над первой частью таблицы, над другими частями с абзацного отступа пишут «Продолжение таблицы» и указывают ее номер, например, «Продолжение таблицы 1». При переносе таблицы на другую страницу заголовок помещают только над ее первой частью.

Таблицы, за исключением таблиц приложений, следует нумеровать арабскими цифрами сквозной нумерацией. Допускается нумерация таблиц в пределах главы. В этом случае номер таблицы состоит из номера главы и порядкового номера таблицы, разделенных точкой, например, таблица 1.2.

Таблицы каждого приложения обозначают отдельной нумерацией арабскими цифрами с добавлением перед цифрой буквенного обозначения приложения, например, «Таблица В.1».

Заголовки граф (столбцов) и строк таблицы следует писать с прописной буквы в единственном числе, а подзаголовки граф – со строчной буквы, если они составляют одно предложение с заголовком, или с прописной буквы, если они имеют самостоятельное значение. В конце заголовков и подзаголовков таблиц точки не ставятся. Заголовки граф, как правило, записываются параллельно строкам таблицы. При необходимости допускается перпендикулярное расположение заголовков граф.

Таблицы располагают только после текста, в котором идет о них речь, или на следующей странице. Ни в рисунках, ни в таблицах не должно быть элементов, о которых не идет речь в дипломной работе.

Формулы выводятся на свободные строки (по центру) и могут сопровождаться экспликацией, отделяемой от формулы запятой. В экспликации разъясняется смысл величин и коэффициентов в той последовательности, как они стоят в формуле. Первая строка экспликации начинается словом «где» (двоеточие после него не ставится), затем обозначение первой величины и через тире его расшифровка. После каждой расшифровки ставится точка с запятой, а после последней – точка.

Пример:

 , (2.6)

где (x_0, y_0, z_0) - координаты точки на плоскости.

Номер формулы, заключенный в круглые скобки, ставится на последней строке формулы и обозначается двумя числами – номером главы и порядковым номером в данной главе. Например, «(2.5)» - вторая глава, пятая формула. Если же в ВКР принята сквозная нумерация формул, то в круглых скобках отмечается очередной номер, например, (32). Формула сопровождается номером только в случае, если на нее имеется ссылка в последующем тексте.

Если формула не умещается в одну строку, то она должна быть перенесена после математического знака на другую строку с повторением этого знака в следующей строке.

Формулы, помещаемые в приложениях, должны нумероваться отдельной нумерацией арабскими цифрами в пределах каждого приложения с добавлением перед каждой цифрой обозначения приложения, например, формула «(В.1)». Ссылки в тексте на порядковые номера формул дают в скобках, например, «… в формуле (1)».

Математические знаки "+", "–", ">", "<" используются только в формулах, таблицах и рисунках. В тексте данные знаки должны быть обозначены словами "плюс", "минус", "больше", "меньше" и так далее. Если в тексте приводится диапазон изменений какой-либо величины, то обозначение единиц указывается только после последнего диапазона, например, "... отклонения величин лежат в диапазоне от 8 до 12%... ", или "... отклонения величин лежат в диапазоне 8–12%...". Не допускается отделять единицу величины от числового значения (переносить ее на другую строку или другую страницу).

Между последней цифрой числа и обозначением единицы следует оставлять пробел.

Исключения составляют обозначения в виде знака, поднятого над строкой, например 80%, 20˚ и так далее. Единица величины одного и того же параметра в пределах всей работы должна быть постоянной.

Оформление ссылок на литературные источники

Цитаты (выдержки) из источников и литературы используются в тех случаях, когда свою мысль хотят подтвердить точной выдержкой по определенному вопросу. Цитаты должны быть текстуально точными и заключены в кавычки. Если в цитату берется часть текста, то есть не с начала фразы или с пропусками внутри цитируемой части, то место пропуска обозначается отточиями (тремя точками). В тексте необходимо указать в квадратных скобках порядковый номер источника в списке литературы и номер процитированной страницы. Например: [5, 236]. Так делается в случае дословного цитирования. Если же просто ссылаются на соответствующее место в источнике, то перед его номером ставится «См.:». Например: [См.: 11, 118]. При ссылке на источник в конце предложения перед точкой ставится его порядковый номер в квадратных скобках [5], если ссылка содержит несколько источников, то они указываются через запятую [1, 98].

Защита курсовой работы

Защита курсовой работы осуществляется студентом публично перед комиссией из числа представителей профессорско-преподавательского состава кафедры в рамках установленного кафедрой графика. Важными условиями допуска курсовой работы к защите является предоставление чистового текста курсовой работы научному руководителю за две недели до установленной даты защиты, а также наличие отзыва научного руководителя. В котором дается объективная характеристика курсовой работы, отмечаются ее достоинства и недостатки, рекомендованная оценка к защите.. Научный руководитель должен присутствовать на защите курсовой работы.

При выставлении итоговой оценки за курсовую работу для комиссии основополагающими являются указанные выше критерии оценки, а также уровень знаний студента по исследуемой теме и по базовой дисциплине в целом, наличие творческого подхода и самостоятельности в исследовании. Члены комиссии, учитывая мнение научного руководителя, а также общее соответствие работы и защиты критериям, выставляют итоговую оценку за курсовую работу, научный руководитель в тот же день выставляет ее в ведомость и зачетную книжку студента.

10. ПЕРЕЧЕНЬ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, ИСПОЛЬЗУЕМЫХ ПРИ ОСУЩЕСТВЛЕНИИ ОБРАЗОВАТЕЛЬНОГО ПРОЦЕССА ПО ДИСЦИПЛИНЕ

1. Microsoft Imagine Premium: Microsoft Windows Professional 7;
2. Microsoft Office 2007 Russian Academic OPEN No Level;
3. Антивирусное программное обеспечение Kaspersky Security.
4. ОС Linux
5. GNU gcc

11. ОПИСАНИЕ МАТЕРИАЛЬНО-ТЕХНИЧЕСКОЙ БАЗЫ, НЕОБХОДИМОЙ ДЛЯ ОСУЩЕСТВЛЕНИЯ ОБРАЗОВАТЕЛЬНОГО ПРОЦЕССА ПО ДИСЦИПЛИНЕ

Вид занятия	***Наименование аудитории***	***Перечень основного оборудования***	***№ ЛР***
1	**2**	**3**	**4**
Лк	Лекционная аудитория	-	-
ЛР	Лаборатория параллельных вычислений	Персональные компьютеры i5-2500/H67/4Gb/500Gb (монитор TFT19 Samsung E1920NR); интерактивная доска Smart	№ 1-9

		Board X885ix со встроенным проектором UX60	
КР	Лаборатория параллельных вычислений	Персональные компьютеры i5-2500/H67/4Gb/500Gb (монитор TFT19 Samsung E1920NR);	-
СР	Ч31	Оборудование 10 ПК i5-2500/H67/4Gb(монитор TFT19 Samsung); принтер HP LaserJet P2055D	-

Приложение 1

ФОНД ОЦЕНОЧНЫХ СРЕДСТВ ДЛЯ ПРОВЕДЕНИЯ ПРОМЕЖУТОЧНОЙ АТТЕСТАЦИИ ОБУЧАЮЩИХСЯ ПО ДИСЦИПЛИНЕ

1. Описание фонда оценочных средств (паспорт)

№ компетенции	Элемент компетенции	Раздел	Тема	ФОС
ОПК-3	Способность к разработке алгоритмических и программных решений в области системного и прикладного программирования, математических, информационных и имитационных моделей,	**1**. Общие сведения об операционных системах и их структурных элементах	1.1. Архитектура операционных систем	Индивидуальное задание, экзаменационный вопрос
			1.2. Файлы и каталоги	Индивидуальное задание, экзаменационный вопрос
ПК-1	Способность собирать, обрабатывать и интерпретировать данные современных научных исследований, необходимые для формирования выводов по соответствующим научным исследованиям		1.3. Понятие процесса в системе	Индивидуальное задание, экзаменационный вопрос
			1.4. Реализация многозадачности.	Индивидуальное задание, экзаменационный вопрос
			1.5. Сигналы	Индивидуальное задание, экзаменационный вопрос
ПК-7	Способность к разработке и применению алгоритмических и программных решений в области системного и прикладного программного обеспечения	**2.** Межпроцессное взаимодействие	2.1 Каналы	Индивидуальное задание, экзаменационный вопрос
			2.2 Проблемы межпроцессного взаимодействия	Индивидуальное задание, экзаменационный вопрос
			2.3. Синхронизация	Индивидуальное задание, экзаменационный вопрос
ПК-9	Способность составлять и контролировать план выполняемой работы, планировать необходимые для выполнения работы ресурсы, оценивать результаты собственной работы		2.4. Управление памятью	Индивидуальное задание, экзаменационный вопрос
			2.5. Разделяемая память	Индивидуальное задание, экзаменационный вопрос

2. Экзаменационные вопросы

№ п/п	Компетенции		ЭКЗАМЕНАЦИОННЫЕ ВОПРОСЫ	№ и наименование раздела
	Код	Определение		
1	2	3	4	5
1.	ОПК-3	Способность к разработке алгоритмических и программных решений в области системного и прикладного программирования, математических, информационных и имитационных моделей,	1. Ядро операционной системы. Типы архитектур ядер операционных систем.	**1**. Общие сведения об операционных системах и их структурных элементах
			2. Ядро Linux и его особенности . Специфические особенности программирования работы ядра в сравнении с пользовательской программой.	**1**. Общие сведения об операционных системах и их структурных элементах
			3. Абстракция файла. Режим файла и режим доступа.	**1**. Общие сведения об операционных системах и их структурных элементах
			4. Каталоги. Операции с каталогами. Узлы устройств.	**1**. Общие сведения об операционных системах и их структурных элементах
2.	ПК-1	Способность собирать, обрабатывать и интерпретировать данные современных научных исследований, необходимые для формирования выводов по соответствующим научным исследованиям	5. Программы, процессы и потоки. Атрибуты процесса. Типы процессов. Состояние процесса.	**1**. Общие сведения об операционных системах и их структурных элементах
			6. Порождение нового процесса fork(). Передача управления процессу: семейство функций exec*().	**1**. Общие сведения об операционных системах и их структурных элементах
			7. Управление процессами в системе. Системы с кооперативной многозадачностью и системы с преемптивной многозадачностью.	**1**. Общие сведения об операционных системах и их структурных элементах
			8. Основные алгоритмы планирования.	**1**. Общие сведения об операционных системах и их структурных элементах
			9. Виды сигналов. Порождение сигнала.	**1**. Общие сведения об операционных системах и их структурных элементах
			10. Перехват сигнала. Обработчики сигнала.	**1**. Общие сведения об операционных системах и их структурных элементах
3.	ПК-7	Способность к разработке и применению алгоритмических и программных решений в области системного и прикладного программного	11. Неименованные каналы. Системный вызов pipe(). Чтение и запись в неименованный канал.	**2.** Межпроцессное взаимодействие
			12. Файлы FIFO. Создание файла. Работа с FIFO.	**2.** Межпроцессное взаимодействие
			13. Конкуренция процессов за ресурсы. Критические области.	**2.** Межпроцессное взаимодействие
			14. Тупиковые состояния. Модели взаимоблокировок.	**2.** Межпроцессное взаимодействие

		обеспечения	15. Средства синхронизации.	2. Межпроцессное взаимодействие
4.	ПК-9	Способность составлять и контролировать план выполняемой работы, планировать необходимые для выполнения работы ресурсы, оценивать результаты собственной работы	16. Адресное пространство процесса. Управление динамической памятью приложения.	2. Межпроцессное взаимодействие
			17. Организация памяти в системе. Интерфейсы для работы с памятью в ядре.	2. Межпроцессное взаимодействие
			18. Отображение файла на память. Анонимные отображения в памяти.	2. Межпроцессное взаимодействие
			19. Использование разделяемой памяти. Блокировка участка памяти.	2. Межпроцессное взаимодействие

3. Описание показателей и критериев оценивания компетенций

Показатели	Оценка	Критерии
Знать (*ОПК-3*): - основные алгоритмы решения задач (*ПК-1*): - способы сбора и обработки информации; (*ПК-7*): - принципы построения программных решений ; (*ПК-9*): - принципы планирования работы по осуществлению разработки системных программ; **Уметь** (*ОПК-3*): - разрабатывать алгоритмы решения задач из области системного и прикладного программирования, математических, информационных и имитационных моделей ; (*ПК-1*):	**отлично**	Демонстрирует все показатели компетенций на высоком уровне, а именно: -знает основные алгоритмы решения задач; -знает способы сбора и обработки информации; -знает принципы построения программных решений ; - знает принципы планирования работы по осуществлению разработки системных программ -умеет разрабатывать алгоритмы решения задач из области системного и прикладного программирования, математических, информационных и имитационных моделей ; -умеет применять аппарат математической статистики для обработки данных; -умеет разрабатывать системное и прикладного программное обеспечение; - умеет разрабатывать, оценивать и реализовывать процессы жизненного цикла программного обеспечения; -владеет приемами построения алгоритмических и программных решений; -владеет навыками разработки программных решений - владеет навыками планирования, контроля и оценки результатов собственной деятельности.
	хорошо	Демонстрирует освоенность не менее 7 показателей компетенций: -знает основные алгоритмы решения задач; -знает способы сбора и обработки информации; -знает принципы построения программных

- применять аппарат математической статистики для обработки данных; (*ПК-7*): - разрабатывать системное и приклад-ное программное обеспечение (*ПК-9*): – разрабатывать, оценивать и реализовывать процессы жизненного цикла программного обеспечения; **Владеть:** (*ОПК-3*): - приемами построения алгоритмических и программных решений. (*ПК-1*): – методами и приемами обработки данных и интерпретации результатов (*ПК-7*): – навыками разработки программных решений; (*ПК-9*): - навыками планирования, контроля и оценки результатов собственной деятельности.		решений ; -умеет разрабатывать алгоритмы решения задач из области системного и прикладного программирования, математических, информационных и имитационных моделей ; -умеет применять аппарат математической статистики для обработки данных; -умеет разрабатывать системное и прикладного программное обеспечение; -владеет приемами построения алгоритмических и программных решений; - владеет навыками планирования, контроля и оценки результатов собственной деятельности; или -знает основные алгоритмы решения задач; -знает способы сбора и обработки информации; -знает принципы построения программных решений ; -умеет разрабатывать алгоритмы решения задач из области системного и прикладного программирования, математических, информационных и имитационных моделей ; -умеет применять аппарат математической статистики для обработки данных; -умеет разрабатывать системное и прикладного программное обеспечение; -владеет методами и приемами обработки данных и интерпретации результатов; - владеет навыками планирования, контроля и оценки результатов собственной деятельности
	удовлетворительно	Демонстрирует владение не менее, чем половиной компетенций: -знает основные алгоритмы решения задач; -знает способы сбора и обработки информации; -знает принципы построения программных решений ; -умеет разрабатывать алгоритмы решения задач из области системного и прикладного программирования, математических, информационных и имитационных моделей ; -умеет применять аппарат математической статистики для обработки данных.
	неудовлетворительно	Владеет менее чем половиной параметров компетенций.

4. Методические материалы, определяющие процедуры оценивания знаний, умений, навыков и опыта деятельности

Дисциплина Системное программирование направлена на ознакомление обучающихся с основами разработки системных программ, способами использования и управления системными ресурсами; на получение теоретических знаний и практических навыков разработки программ, формулирования и решения проблем из различных областей наук, а также осуществления поиска, хранения, обработки и анализа информации из различных источников и представления ее в соответствующем виде и для их дальнейшего использования в практической деятельности.

Изучение дисциплины Системное программирование предусматривает:

- лекции,
- лабораторные работы;
- курсовую работу;
- экзамен;
- самостоятельную работу.

Для фиксирования успешности обучения предусматривается экзамен.

В ходе освоения раздела 1 «Общие сведения об операционных системах и их структурных элементах» обучающиеся должны изучить принципы устройства и функционирования операционных систем, их структуру, способы обращения к компонентам системы, научиться получать информацию о процессах в системе, атрибутах файлов, использовать системные ресурсы в прикладных и системных программах.

В ходе освоения раздела 2 «Межпроцессное взаимодействие» обучающиеся осваивают особенности управления процессами в системе, использования механизмов передачи сообщений, отправку и перехват сигналов.

Студентам необходимо овладеть навыками и умениями применения изученных методов для разработки и реализации профессионально ориентированных проектов в последующей учебной деятельности.

Овладение ключевыми понятиями является основой усвоения учебного материала по дисциплине.

При подготовке к экзамену особое внимание необходимо уделить рекомендациям и замечаниям преподавателей, ведущих аудиторные занятия по дисциплине

В процессе проведения лабораторных занятий происходит закрепление знаний, формирование умений и навыков применения различных методов решения стандартных математических ситуаций.

Самостоятельную работу необходимо начинать с чтения лекций и учебников.

В процессе консультации с преподавателем обучающийся выясняет наличие пробелов в знаниях и способах решения разных ситуаций.

Работа с литературой является важнейшим элементом в получении знаний по дисциплине. Прежде всего, необходимо воспользоваться списком рекомендуемой по данной дисциплине литературой. Дополнительные сведения по изучаемым темам можно найти в периодической печати и Интернете.

Предусмотрено проведение аудиторных занятий в виде разнообразных тренингов и ситуаций общения в сочетании с внеаудиторной работой.

Приложение 2

АННОТАЦИЯ
рабочей программы дисциплины
Системное программирование

1. Цель и задачи дисциплины

Целью изучения дисциплины является: ознакомление обучающихся с различными методами, приемами разработки системных программ, приемами интеграции одних программных пакетов в другие и использованию результатов интеграции при создании собственных сложных универсальных программных комплексов.

Задачами дисциплины являются

- изучение механизмов и алгоритмов функционирования операционных систем и их компонентов;
- освоение приемов использования системных ресурсов при разработке системных и прикладных программ;
- приобретение навыков написания программ, отвечающим требованиям безопасности.

2. Структура дисциплины

2.1 Распределение трудоемкости по отдельным видам учебных занятий, включая самостоятельную работу: Лк.- 17 час., ЛР- 51 час.; СР - 40 час.

Общая трудоемкость дисциплины составляет 144 часов, 4 зачетных единиц.

2.2 Основные разделы дисциплины:

1 – Общие сведения об операционных системах и их структурных элементах;
2 – Межпроцессное взаимодействие.

3. Планируемые результаты обучения (перечень компетенций)

Процесс изучения дисциплины направлен на формирование следующих компетенций:

ОПК-3 Способность к разработке алгоритмических и программных решений в области системного и прикладного программирования, математических, информационных и имитационных моделей, созданию информационных ресурсов глобальных сетей, образовательного контента, прикладных баз данных, тестов и средств тестирования систем и средств на соответствие стандартам и исходным требованиям.
ПК-1 Способность собирать, обрабатывать и интерпретировать данные современных научных исследований, необходимые для формирования выводов по соответствующим научным исследованиям .
ПК-7 Способность к разработке и применению алгоритмических и программных решений в области системного и прикладного программного обеспечения .
ПК-9 Способность составлять и контролировать план выполняемой работы, планировать необходимые для выполнения работы ресурсы, оценивать результаты собственной работы

4. Вид промежуточной аттестации: экзамен.

Приложение 3

Протокол о дополнениях и изменениях в рабочей программе
на 20___-20___ учебный год

1. В рабочую программу по дисциплине вносятся следующие дополнения:

_____
_____

2. В рабочую программу по дисциплине вносятся следующие изменения:

_____
_____
_____

Протокол заседания кафедры №____ от «___» _____ 20 ____ г.,
(разработчик)

Заведующий кафедрой _____ _____
(подпись) *(Ф.И.О.)*

Приложение 4

ФОНД ОЦЕНОЧНЫХ СРЕДСТВ ДЛЯ ТЕКУЩЕГО КОНТРОЛЯ УСПЕВАЕМОСТИ ПО ДИСЦИПЛИНЕ

1. Описание фонда оценочных средств (паспорт)

№ компетенции	Элемент компетенции	Раздел	Тема	ФОС
ОПК-3	Способность к разработке алгоритмических и программных решений в области системного программирования	**1.** Общие сведения об операционных системах и их структурных элементах	1.1. Архитектура операционных систем	КР
			1.2. Файлы и каталоги	ЛР№ 1, 2, КР
ПК-1	Способность собирать, обрабатывать и интерпретировать данные современных научных исследований, необходимые для формирования выводов по соответствующим научным исследованиям		1.3. Понятие процесса в системе	ЛР№3, КР
			1.4. Реализация многозадачности.	ЛР№ 4
			1.5. Сигналы	ЛР№5
ПК-7	Способность к разработке и применению алгоритмических и программных решений в области системного и прикладного программного обеспечения	**2.** Межпроцессное взаимодействие	2.1 Каналы	ЛР№ 6
			2.2 Проблемы межпроцессного взаимодействия	КР
ПК-9	Способность составлять и контролировать план выполняемой работы, планировать необходимые для выполнения работы ресурсы, оценивать результаты собственной работы		2.3. Синхронизация	ЛР№ 7
			2.4. Управление памятью	ЛР№18
			2.5. Разделяемая память	ЛР№ 9

2. Описание показателей и критериев оценивания компетенций

Показатели	Оценка	Критерии
(*ОПК-3*): - основные алгоритмы решения задач (*ПК-1*): - способы сбора и обработки информации; (*ПК-7*): - принципы построения программных решений ; (*ПК-9*): - принципы планирования работы по	**отлично**	Демонстрирует все показатели компетенций на высоком уровне, а именно: -знает основные алгоритмы решения задач; -знает способы сбора и обработки информации; -знает принципы построения программных решений ; - знает принципы планирования работы по осуществлению разработки системных программ; -умеет разрабатывать алгоритмы решения задач из области системного и прикладного программирования, математических, информационных и имитационных моделей ;

осуществлению разработки системных программ; **Уметь** (*ОПК-3*): - разрабатывать алгоритмы решения задач из области системного и прикладного программирования, математических, информационных и имитационных моделей ; (*ПК-1*): - применять аппарат математической статистики для обработки данных; (*ПК-7*): - разрабатывать системное и приклад-ное программное обеспечение (*ПК-9*): – разрабатывать, оценивать и реализовывать процессы жизненного цикла программного обеспечения; **Владеть:** (*ОПК-3*): - приемами построения алгоритмических и программных решений. (*ПК-1*): – методами и приемами обработки данных и интерпретации результатов (*ПК-7*): – навыками разработки программных решений; (*ПК-9*): - навыками планирования, контроля и оценки результатов собственной деятельности.		-умеет применять аппарат математической статистики для обработки данных; -умеет разрабатывать системное и прикладного программное обеспечение; -умеет разрабатывать, оценивать и реализовывать процессы жизненного цикла программного обеспечения; -владеет приемами построения алгоритмических и программных решений; -владеет навыками разработки программных решений.
	хорошо	Знает основные алгоритмы решения задач; -знает способы сбора и обработки информации; -знает принципы построения программных решений ; -умеет разрабатывать алгоритмы решения задач из области системного и прикладного программирования, математических, информационных и имитационных моделей ; -умеет применять аппарат математической статистики для обработки данных; -умеет разрабатывать системное и прикладного программное обеспечение; -владеет методами и приемами обработки данных и интерпретации результатов; - владеет навыками планирования, контроля и оценки результатов собственной деятельности.
	удовлетвори тельно	-знает основные алгоритмы решения задач; -знает способы сбора и обработки информации; -знает принципы построения программных решений ; -умеет разрабатывать алгоритмы решения задач из области системного и прикладного программирования, математических, информационных и имитационных моделей ; -владеет приемами построения алгоритмических и программных решений.
	неудовлетво рительно	Владеет менее чем половиной параметров компетенций.

Программа составлена в соответствии с федеральным государственным образовательным стандартом высшего образования по направлению подготовки 01.03.02 Прикладная математика и информатика от «12» марта 2015 г. № 228

для набора 2015 года: и учебным планом ФГБОУ ВО «БрГУ» для очной формы обучения от «13 » июля 2015 г. № 475

для набора 2016 года: и учебным планом ФГБОУ ВО «БрГУ» для очной формы обучения от «06» июня 2016г. № 429

для набора 2017 года: и учебным планом ФГБОУ ВО «БрГУ» для очной формы обучения от «6» марта 2017г. № 125

для набора 2018 года и учебным планом ФГБОУ ВО «БрГУ» для очной формы обучения от «12» марта 2018г. №130

Программу составил:

Ратинская Е.В., ст. препод. каф. МиФ _____

Рабочая программа рассмотрена и утверждена на заседании кафедры МиФ

от «21» ноября 2018 г., протокол № 3

И.о. зав.выпускающей кафедрой _____ О.И.Медведева

СОГЛАСОВАНО:

И.о. зав.выпускающей кафедрой _____ О.И Медведева.

Директор библиотеки _____ Т.Ф.Сотник

Рабочая программа одобрена методической комиссией ЕН факультета

от «20 » декабря 2018 г., протокол № 4

Председатель методической комиссии факультета _____ М.А. Варданян

СОГЛАСОВАНО:

Начальник
учебно-методического управления _____ Г.П. Нежевец

Регистрационный №_____