МАТЕМАТИЧЕСКОЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ, КОМПЛЕКСОВ И КОМПЬЮТЕРНЫХ СЕТЕЙ / MATHEMATICAL SOFTWARE FOR COMPUTERS, COMPLEXES AND COMPUTER NETWORKS

DOI: https://doi.org/10.23670/IRJ.2023.138.119

РАЗРАБОТКА УЧЕБНОЙ ОПЕРАЦИОННОЙ СИСТЕМЫ

Научная статья

Белоус В.С.[1], Белоус Н.Н.[2], Шубабко Е.Н.[3, *]
[2] ORCID : 0009-0001-3118-8881;
[3] ORCID : 0009-0008-1683-2203;
[1] Mail.ru Group, Москва, Российская Федерация
[2, 3] Брянский государственный университет, Брянск, Российская Федерация

* Корреспондирующий автор (elenashubabko[at]gmail.com)

Аннотация

Данная статья посвящена вопросам разработки и функционирования учебной операционной системы. Студенты, осваивающие программную инженерию или изучающие системное программное обеспечение, изучают функционирование современных операционных систем. Однако будущие разработчики должны кроме этого разбираться в исходном коде и уметь его модифицировать. Для этого в учебном процессе целесообразно использовать более простые операционные системы, имеющие сокращенный функционал, доступные для модификации и усложнения студентами. В статье представлены результаты разработки такой операционной системы, а именно – описан применяемый нами подход к решению поставленной задачи и основной результат – сборка ядра предлагаемой операционной системы.

Ключевые слова: операционная система, ядро, сборка, прерывания, исключения.

DEVELOPMENT OF A TRAINING OPERATING SYSTEM

Research article

Belous V.S.[1], Belous N.N.[2], Shubabko E.N.[3, *]
[2] ORCID : 0009-0001-3118-8881;
[3] ORCID : 0009-0008-1683-2203;
[1] Mail.ru Group, Moscow, Russian Federation
[2, 3] Bryansk State University, Bryansk, Russian Federation

* Corresponding author (elenashubabko[at]gmail.com)

Abstract

This article is dedicated to the issues of development and functioning of a training operating system. Students mastering software engineering or studying system software learn the functioning of modern operating systems. However, future developers should besides understand the source code and be able to modify it. For this purpose it is reasonable to use simpler operating systems with reduced functionality available for modification and complication by students in the educational process. This article presents the results of the development of such an operating system, namely, it describes the approach we used to solve the task and the main result – building the core of the proposed operating system.

Keywords: operating system, core, build, interrupts, exceptions.

Введение

Для изучения студентами – будущими программистами основ функционирования операционных систем (далее – ОС) недостаточно освоения только теоретического материала. Для понимания работы ядра ОС необходимо изучать и модифицировать его исходный код. Вопросам разработки и функционирования ОС посвящены классические работы [1], [2].

В настоящее время существует множество операционных систем с открытым исходным кодом: GNU Linux, FreeBSD, ReactOS и др. Однако, ядра этих операционных систем плохо подходят для учебного процесса, поскольку они имеют большой объем исходного кода, который обладает высокой сложностью. По этой причине были разработаны несколько учебных операционных систем: JOS [3], xv6 [4] и PhantomEx [5]. Ядра этих ОС имеют небольшой объем исходного кода и небольшую сложность по сравнению с эксплуатируемыми ОС, что делает их пригодными для обучения студентов системному программированию [6].

Однако данные ОС имеют один существенный недостаток: они разработаны под устаревшую архитектуру x86. Поэтому разработка учебной операционной системы для более современной архитектуры AMD64 является актуальной задачей.

Методы и принципы разработки

Разработка и отладка ОС производилась с использованием системы Gentoo 2.2 (ядро GNU Linux 4.1.15). Для сборки ядра использовались: GNU Make 4.1, gcc 4.9.3, GNU Binutils 2.25, GNU Perl 5.20.2, GNU gdb 7.10.1.

Разработка ОС выполнялась на языке Си с ассемблерными вставками. В ассемблерных вставках используется синтаксис AT&T, т.к. он является стандартным для средств разработки GNU и большинства Unix-подобных систем.

Существует 2 альтернативных синтаксиса ассемблера: AT&T и Intel. Их основные различия заключаются в следующем.

Порядок операндов. В синтаксисе AT&T присваивание выполняется слева направо (**movl $1, %eax**). В Intel – наоборот, справа налево (**moveax, 1**).

Суффиксы размерностей операндов. В AT&T команды заканчиваются однобуквенным суффиксом, показывающим разрядность операндов:

q – quad, восемь байт;

l – long, четыре байта;

w – word, два байта;

b – byte, один байт.

В случае Intel – для обозначения размер операндов используются префиксы: **byte ptr, word ptr** и т.д.

Префиксы операндов. В AT&T операнд дополняется особым символом – префиксом, указывающим его вид:

— регистр – процентом: **%eax**;

— непосредственный операнд – символом USD: **$1**;

— косвенный адрес перехода – звездочкой: ***addr**.

Прочие операнды не имеют префикса: **0x10000** (содержимое памяти, по адресу 0x10000), **varname**. В AT&T **varname** всегда обозначает ячейку памяти, а **$varname** – ее адрес.

Адресация база-масштаб-смещение. В синтаксисе Intel сложение базы со смещением имеет вид: **[var+eax+2*ebx]**. В синтаксисе AT&T используется форма **var(%eax,2,%ebx)**. В случае, если какая-то из частей адреса отсутствует, она пропускается: **-4(,%eax,2)** есть **[2*eax-4]** в нотации Intel.

Тестовое окружение. При разработке операционной системы, как и при разработке любого другого ПО, необходима возможность запуска и проверки работоспособности внесенных изменений. Существует два основных способа запуска ОС: на реальной машине и на виртуальной машине.

К достоинствам использования виртуальной машины можно отнести: возможность запуска ОС сразу после компиляции, без необходимости синхронизации изменений между несколькими компьютерами. Кроме того, некоторые виртуальные машины выводят отладочную информацию, которая может быть полезна при отладке. Основной недостаток использования виртуальных машин заключается в том, что они поддерживают ограниченный набор устройств и могут содержать ошибки.

Основным достоинством тестирования на реальной машине является то, что если ОС работает на одной машине – с большой вероятностью она будет работать и на других машинах такой же архитектуры. В то же время, то, что ОС работает на виртуальной машине не гарантирует, что она будет работать и на реальной. Однако, для тестирования на реальной машине необходимо либо тестировать на той же машине, на которой ведется разработка (что отнимает много времени, т.к. требуются постоянные перезагрузки), либо использовать вторую такую же машину, что требует дополнительных расходов.

Поэтому использование виртуальной машины является предпочтительным.

В работе нами используется эмулятор QEMU 2.5.0, т.к. он имеет поддержку GDB, что позволяет выполнять отладку на уровне исходных кодов. Кроме того, данный эмулятор имеет встроенный монитор, который позволяет переключаться между различными CPU, выводить содержимое регистров, кеша TLB, таблиц страничного преобразования и других системных структур данных.

Организация исходных кодов. Исходные файлы ОС разделены по следующим каталогам:

— user – файлы режима пользователя.

— stdlib – файлы стандартной библиотеки, общей для ядра и прикладных программ.

— stdlib/i386 – место размещения файлов, полученных в ходе компиляции 32-битной версии библиотеки.

— stdlib/x86_64 – место размещения файлов, полученных в ходе компиляции 64-битной версии библиотеки.

— kernel – основные файлы ядра ОС.

— kernel/boot – файлы первого загрузчика ОС.

— kernel/loader – файлы второго загрузчика ОС.

— kernel/interrupt – файлы, связанные с обработкой прерываний.

— kernel/lib/console – файлы, связанные с выводом на экран.

— kernel/lib/disk – файлы, связанные с доступом к диску.

— kernel/lib/memory – файлы, связанные с работой с памятью.

— kernel/lib/i386 – место размещения файлов, полученных в ходе компиляции 32-битной версии библиотеки ядра.

— kernel/lib/x86_64 – место размещения файлов, полученных в ходе компиляции 64-битной версии библиотеки ядра.

— kernel/misc – заголовочные файлы, не подходящие под указанные выше категории.

Исполняемые форматы. В качестве формата исполняемых файлов был выбран ELF – стандартный формат исполняемых файлов для большинства Unix-подобных систем. Преимуществами данного формата является то, что он хорошо документирован и абсолютное большинство Unix-подобных систем имеют компилятор и компоновщик для создания ELF-файлов.

Для упрощения работы с ELF-файлами были разработаны макросы:

ELF32_PHEADER_FIRST, ELF32_PHEADER_LAST, ELF64_PHEADER_FIRST и ELF64_PHEADER_LAST. Пример использования макросов для загрузки второго загрузчика ОС, показан в листинге 1.

Листинг 1 — Пример использования макросов для работы с ELF-файлами:

```
1 for (struct elf32_program_header *ph = ELF32_PHEADER_FIRST(elf_header);
2 ph < ELF32_PHEADER_LAST(elf_header ) ; ph++) {
```

```
3 . . .
4 }
```

Основные результаты

3.1. Сборка ядра

Для запуска ядра в эмуляторе необходимо подготовить образ диска, который включает в себя бинарный файл первого загрузчика, бинарный файл второго загрузчика и бинарный файлы ядра.

После того как эти файлы собраны, образ диска генерируется с помощью утилиты dd, в листинге 2 показано правило, используемое для его генерации.

Листинг 2 – Правило для генерации образа диска:

```
1 ${IMAGE}: $(KERNEL) $(BOOTLOADER) $(LOADER)
2 dd if=/dev/zero of=${IMAGE} bs=1M count=10
3 dd if=$ (BOOTLOADER) of=${IMAGE} conv=notrunc
4 dd if=$ (LOADER) of=${IMAGE} s e ek=1 conv=notrunc
5 dd if=$ (KERNEL) of=${IMAGE} bs=1M seek=1 conv=notrunc
```

3.1.1. Сборка первого загрузчика

Основные фрагменты мейкфайла для сборки загрузчика представлены в листинге 3.

Листинг 3 – Фрагмент мейкфайла для сборки первого загрузчика:

```
1 BOOTLOADER_LDFLAGS = ${LDFLAGS} ${X86_LDFLAGS} \
2 −Wl,−−entry=boot_entry −Wl,−Ttext −Wl, 0x7c00
3
4 $(BOOTLOADER) : ${ bootloader_objects }
5 $(CC) −o $@ ${BOOTLOADER_LDFLAGS} $^
6 $(OBJCOPY) −−only−keep−debug $@ $@.debug
7 $(OBJCOPY) −S −O binary −j.text $@ $@.strip
8 $(PERL) ${BOOTLOADER_DIR}/ sign.pl $@.strip
9 mv $@.strip $@
```

Следует обратить внимание на опции компоновщика: -Wl,–entry=boot_entry -Wl,-Ttext -Wl,0x7c00, которые сообщают компоновщику какой символ является точкой входа в загрузчик (boot_entry) и по какому адресу он должен быть расположен (0x7c00).

После сборки загрузчик имеет размер больше 510 байт, т.к. содержит отладочные символы, которые необходимы для выполнения отладки на уровне исходных кодов, поэтому их необходимо сохранить в отдельный файл с помощью утилиты objcopy, после чего их можно удалить из основного файла с помощью той же утилиты.

Заключительным этапом подготовки загрузчика является создание сигнатуры, чтобы BIOS определил, что первый сектор является загрузочным, для этого необходимо чтобы байты 510 и 511 загрузчика (если считать с 0) были равны 0x55 и 0xAA соответственно). Для этого используется скрипт sign.pl, фрагмент которого приведен в листинге 4.

Листинг 4 — Фрагмент скрипта для создания сигнатуры:

```
1 my $filename = shift or die "Usage: $0 <FILE>\n" ;
2 open my $fh, '>>', $filename
3 or die " can't open ' $filename ' : $!\ n" ;
4
5 print {$fh} "\0" x (510 − $size ) ;
6 print {$fh} "\x55\xAA" ;
7
8 close $fh
```

3.1.2. Сборка второго загрузчика

Ядро ОС и второй загрузчик используют общий код: управление памятью, вывод на экран, работа с диском. Однако, ядро является 64-битным, а загрузчик – 32-битным, поэтому необходимо иметь возможность собирать библиотеки под разные архитектуры: x86 и x86_64.

Для реализации возможности сборки под разные архитектуры, используется иерархия мейкфайлов, это позволяет включать мейкфайл для сборки библиотеки в мейкфайл верхнего уровня несколько раз, используя различные опции. В листинге 5 приведен фрагмент мейкфайла, используемый для сборки двух версий библиотеки. Результаты сборки сохраняются в директории i386 и x86_64.

Листинг 5 — Фрагмент мейкфайла для сборки библиотек:

```
1 KERNEL_LIB_ARCH := ${ARCH32}
2 KERNEL_LIB_CFLAGS := ${CFLAGS} ${X86_CFLAGS} −O0 \
3 −DKERNEL_BASE=${KERNEL_BASE} −DVADDR_BASE=0
4 include ${KERNEL_LIB_DIR}/Makefile.part
5
6 KERNEL_LIB_ARCH := ${ARCH64}
7 KERNEL_LIB_CFLAGS := ${KERNEL_CFLAGS}
8 include ${KERNEL_LIB_DIR}/Makefile.part
```

При компоновке загрузчика используется скрипт, который задает имя и адрес точки входа, а так же экспортирует символ end, который используется загрузчиком для определения свободной области памяти. В листинге 6 приведены основные фрагменты скрипта компоновки.

Листинг 6 — Фрагменты скрипта компоновки:

```
1 ENTRY(_start)
2 SECTIONS
3 {
4 /* Start from 1MB */
5 . = 1M;
6
7 . text BLOCK(4K) : ALIGN(4K)
8 {
9 * ( . t ext )
10 * ( .long_mode_asm)
11 }
12 . . .
13 PROVIDE( end = . ) ;
14 }
```

Так как ядро не поддерживает файловую систему, прикладные программы были добавлены в образ ядра с помощью компоновщика, используя опцию -Wl,–format=binary -Wl,$USER_PROGS. Данная опция говорит компоновщику о необходимости создать специальные символы: _binary_<путь>_start, _binary_<путь>_end и _binary_<путь>_size для каждого файла из $USER_PROGS. Ядро использует данные символы для загрузки и запуска прикладных программ.

В листинге 7 приведены основные фрагменты мейкфайла для добавления прикладных программ в исполняемый файл ядра ОС.

Листинг 7 — Фрагмент мейкфайла для добавления бинарных файлов:

```
1 USER_DIR = user
2 USER_PROGS = ${USER_DIR}/ hello.bin \
3 ${USER_DIR}/ fork.bin \
4 ${USER_DIR}/ spin.bin \
5 ${USER_DIR}/ exit.bin \
6 ${USER_DIR}/ read_kernel.bin \
7 ${USER_DIR}/read_unmap.bin \
8 ${USER_DIR}/ write_kernel.bin \
9 ${USER_DIR}/write_unmap.bin \
10 ${USER_DIR}/ yield.bin
11
12 KERNEL_LDFALGS = ${LDFLAGS} −O0 −lgcc \
13 -Wl,−−format=binary −Wl,${USER_PROGS} −Wl,−−format=default
```

Подробно реализация загрузчика нашей учебной ОС представлена в статье [7].

3.2. Реализация ядра ОС

3.2.1. Прерывания и исключения

Для возможности обработки исключений и прерываний, необходимо загрузить в IDT, содержащую дескрипторы шлюзов обработчиков прерываний и исключений. В листинге 8 приведен фрагмент кода, выполняющий загрузку IDT.

Листинг 8 — Загрузка IDT:

```
1 struct idtr {
2 uint16_t limit ;
3 void * base ;
4 } __attribute__ ( ( packed ) ) idtr = {
5 sizeof( i d t )−1, i d t
6 } ;
7
8 // Load idt
9 asm volatile ( "lidt %0" :: "m" (idtr));
```

Для объявления точек входа в обработчики прерываний используются 2 макроса, их код приведен в листинге 9.

Листинг 9 — Макросы для объявления точек входа в обработчики прерываний:

```
1 #define interrupt_handler_no_error_code (name , num) \
2 .globl name ; \
3 .type name , @function; \
4 .align 4; \
5 name : \
6 pushq $0 ; /*instead of error code */ \
7 pushq $(num) ; \
8 jmp interrupt_handler_common
9
10 #define interrupt_handler_with_error_code (name , num) \
11 .globl name; \
12 .type name , @function; \
13 .align 4; \
```

```
14 name : \
15 pushq $(num); \
16 jmp interrupt_handler_common
```

Первый макрос используется для объявления точек входа в обработчики прерываний для которых процессор не добавляет в стек код ошибки. Второй – для прерываний с кодом ошибки. Это позволяет использовать одну структу для представления контекста прерывания.

После сохранения номера прерывания все обработчики прерываний используют общий код, который сохраняет в стеке контекст текущего процесса, загружает в сегментные регистры дескрипторы сегментов данных ядра и вызывает обработчик прерываний, написанный на языке Си. Код, выполняющий эти действия, представлен в листинге 10.

Листинг 10 — Общий код обработчиков прерываний:

```
1 interrupt_handler_common :
2 pushq $0x0 // reserve space for segment registers
3 movw %ds, 0(%rsp)
4 movw %es, 2(%rsp)
5 movw %fs, 4(%rsp)
6 movw %gs, 6(%rsp)
7
8 // Save RONs: r15−r8 ,rbp−rax
9 . . .
10
11 pushq %rax
12 movw $GD_KD, %ax
13 movw %ax, %ds
14 movw %ax , %es
15 popq %rax
16
17 // Doesn’t return
18 call interrupt_handler
```

Для возможности обработки прерываний необходимо активировать APIC, в листинге 11 приведен код, который для этого предназначен.

Листинг 11 — Активация локального APIC:

```
1 // 0x1B – msr of local apic
2 // bit 11 − global enable / disable APIC flag
3 asm volatile (
4 "movl $0x1b, %ecx\n\t "
5 "rdmsr\n\t "
6 "btsl $11, %eax\n\t "
7 "wrmsr"
8 );
```

После инициализации локального APIC необходимо настроить перенаправление прерываний (через IO APIC). Для этого был разработан макрос IOAPIC_WRITE, который записывает в регистр, переданный в качестве первого параметра, значение, переданное в качестве второго параметра.

В листинге 12 приведен фрагмент кода перенаправления прерываний от клавиатуры с использованием этого макроса.

Листинг 12 — Настройка IO APIC:

```
1 // keyboard
2 IOAPIC_WRITE(IOREDTBL_BASE+2, INTERRUPT_VECTOR_KEYBOARD) ;
3 IOAPIC_WRITE(IOREDTBL_BASE+3, local_apic_id) ;
```

Для возврата из обработчика прерываний используется функция task_run, которая переводит процесс в состояние TASK_STATE_RUN и восстанавливает контекст процесса, сохраненный обработчиком прерывания. Код данной функции приведен в листинге 13.

Следует обратить внимание, что перед запуском процесса в регистре флагов процесса устанавливается флаг RFLAGS.IF, это выполняется для того, чтобы активировать прерывания при возврате в пространство пользователя, т.к. при передаче управления обработчику прерывания процессор сбрасывает данный флаг.

Листинг 13 — Запуск процесса:

```
1 void task_run( struct task * task ) {
2 // Always enable interrupts
3 task−>context. rflags |= RFLAGS_IF;
4 task−>state = TASK_STATE_RUN;
5
6 asm volatile (
7 "movq %0, %%rsp\n\t "
8
9 // restore gprs: rax−rbp, r8−r15
10 . . .
```

```
11
12 // restore segment registers
13 "movw 0(%%r sp ) , %%ds\n\t "
14 "movw 2(%%r sp ) , %%es\n\t "
15 "addq $0x8 , %%rsp\n\t "
16
17 // skip interrupt_number and error_code
18 "addq $0x10 , %%rsp\n\t "
19
20 " iretq " : : "g" ( task ) : "memory"
21 ) ;
22 }
```

3.2.2. Системные вызовы

Прикладные программы получают доступ к сервисам ядра используя механизм системных вызовов. Для выполнения системного вызова используется код приведенный в листинге 14.

Как видно из листинга, для передачи номера системного вызова используется регистр RAX, этот же регистр используется для последующего возврата результата системного вызова.

Параметры в системный вызов передаются через регистры RBX, RCX, RDX, RDI, RSI [8]. Системные вызовы в разрабатываемой ОС либо не принимают аргументов, либо принимают один аргумент, поэтому 5 регистров достаточно. При реализации системных вызовов, требующих больше 5 аргументов, прикладному ПО необходимо создать на стеке структуру, содержащую требуемые аргументы и передавать ее адрес в одном из регистров. Ядро сможет обращаться к этой структуре, поскольку обработка системного вызова выполняется в контексте процесса, совершившего системный вызов.

Листинг 14 — Выполнение системных вызовов:

```
1 asm volatile ( "int %1\n"
2 : "=a" ( ret )
3 : " i " (INTERRUPT_VECTOR_SYSCALL) ,
4 "a" ( syscall ) ,
5 "b" ( arg1 ) ,
6 " c " ( arg2 ) ,
7 "d" ( arg3 ) ,
8 "D" ( arg4 ) ,
9 "S" ( arg5 )
10 : " cc " , "memory" ) ;
```

Для выполнения системных вызовов прикладные процессы используют следующие функции: sys_puts, sys_yield, sys_exit, sys_fork. В листинге 15 приведен код функции sys_puts, остальные функции реализованы аналогично.

Листинг 15 — Реализация функцииsys_puts:

```
1 void sys_puts( const char * string )
2 {
3 return ( void) syscall (SYSCALL_PUTS, ( uintptr_t ) string , 0 , 0 , 0 , 0) ;
4 }
```

Разработанный программный код всегда нуждается в тестировании и отладке. Для тестирования ОС написаны прикладные программы, которые проверяют различные функции ядра ОС: вывод на экран, чтение и запись данных в различные области памяти, создание и уничтожение процессов, вытесняющая многозадачность. Процесс тестирования и отладки разработанной ОС описан в [9].

Заключение

В результате работы по созданию ядра операционной системы была реализована его основная функция как средства разделения аппаратных ресурсов: физической памяти, процессорного времени и устройства вывода информации. В итоге была создана многозадачная однопользовательская однопроцессорная система с монолитным ядром, по классификации из [1], которую можно использовать в учебном процессе.

Недостатками представленной ОС являются отсутствие файловой системы и поддержки многопроцессорной архитектуры. Тем не менее, созданная система поддерживает архитектуру процессоров AMD64, реализует полноценную изоляцию процессов, виртуальную память на основе страниц, вытесняющую многозадачность, потоки уровня ядра и эффективное клонирование процессов на основе копирования памяти при ее изменении.

Разработанную операционную систему можно улучшить следующим образом: добавить возможность работы с несколькими процессорами и средства синхронизации [10], добавить файловую систему и межпроцессное взаимодействие, добавить поддержку дополнительного аппаратного обеспечения, например – сетевой карты, что можно использовать как задачи для курсовых и дипломных проектов студентов, изучающих программную инженерию.

Конфликт интересов

Не указан.

Рецензия

Сообщество рецензентов Международного научно-исследовательского журнала
DOI: https://doi.org/10.23670/IRJ.2023.138.119.1

Conflict of Interest

None declared.

Review

International Research Journal Reviewers Community
DOI: https://doi.org/10.23670/IRJ.2023.138.119.1

Список литературы / References

1. Tanenbaum A.S. Modern Operating Systems / A.S. Tanenbaum, H. Bos — Upper Saddle River, New Jersey 07458: Prentice Hall PTR, 2014. — 1120 p.
2. Tanenbaum A.S. Operating Systems: Design and Implementation / A.S. Tanenbaum, A. Woodhull — Upper Saddle River, New Jersey 07458: Prentice Hall PTR, 1997. — 576 p.
3. OSDev.org. — 2022 — URL: http://wiki.osdev.org (accessed: 17.10.2023)
4. Cox R. xv6: a Symple, Unix-like Teaching Operating System / R. Cox, F. Kaashoek, R. Morris. — 2014 — URL: https://pdos.csail.mit.edu/6.828/2014/xv6/book-rev8.pdf (accessed: 17.10.2023)
5. PhantomEx // PhantomEx: журнал разработки. — 2014 — URL: http://phantomexos.blogspot.ru (дата обращения: 17.10.2023)
6. Крищенко В.А. Основы реализации операционных систем / В.А. Крищенко // Основы реализации операционных систем . — 2016 — URL: http://mit.spbau.ru/sewiki/images/d/da/Os-dev-jos.pdf (дата обращения: 17.10.2023)
7. Белоус В.С. Реализация загрузчика операционной системы / В.С. Белоус, Н.Н. Белоус // Научно-технический вестник Поволжья. — 2020. — 12. — с. 159-161.
8. Matz M. System V Application Binary Interface AMD64 Architecture Processor Supplement / M. Matz. — 2012 — URL: https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf (accessed: 17.10.2023)
9. Белоус В.С. Тестирование и отладка учебной операционной системы / В.С. Белоус, Н.Н. Белоус, Н.В. Силенок // Научно-технический вестник Поволжья. — 2017. — 1. — с. 69-72.
10. MultiProcessor Specification. — 1997 — URL: https://pdos.csail.mit.edu/6.828/2018/readings/ia32/MPspec.pdf (accessed: 17.10.2023)

Список литературы на английском языке / References in English

1. Tanenbaum A.S. Modern Operating Systems / A.S. Tanenbaum, H. Bos — Upper Saddle River, New Jersey 07458: Prentice Hall PTR, 2014. — 1120 p.
2. Tanenbaum A.S. Operating Systems: Design and Implementation / A.S. Tanenbaum, A. Woodhull — Upper Saddle River, New Jersey 07458: Prentice Hall PTR, 1997. — 576 p.
3. OSDev.org. — 2022 — URL: http://wiki.osdev.org (accessed: 17.10.2023)
4. Cox R. xv6: a Symple, Unix-like Teaching Operating System / R. Cox, F. Kaashoek, R. Morris. — 2014 — URL: https://pdos.csail.mit.edu/6.828/2014/xv6/book-rev8.pdf (accessed: 17.10.2023)
5. PhantomEx [PhantomEx] // PhantomEx: development log. — 2014 — URL: http://phantomexos.blogspot.ru (accessed: 17.10.2023) [in Russian]
6. Krischenko V.A. Osnovy realizatsii operatsionnyh sistem [Fundamentals of Operating Systems Implementation] / V.A. Krischenko // Fundamentals of Operating Systems Implementation. — 2016 — URL: http://mit.spbau.ru/sewiki/images/d/da/Os-dev-jos.pdf (accessed: 17.10.2023) [in Russian]
7. Belous V.S. Realizatsija zagruzchika operatsionnoj sistemy [Implementation of the Operating System Bootloader] / V.S. Belous, N.N. Belous // Scientific and Technical Volga Region Bulletin. — 2020. — 12. — p. 159-161. [in Russian]
8. Matz M. System V Application Binary Interface AMD64 Architecture Processor Supplement / M. Matz. — 2012 — URL: https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf (accessed: 17.10.2023)
9. Belous V.S. Testirovanie i otladka uchebnoj operatsionnoj sistemy [Testing and Debugging of the Training Operating System] / V.S. Belous, N.N. Belous, N.V. Silenok // Scientific and Technical Volga Region Bulletin. — 2017. — 1. — p. 69-72. [in Russian]
10. MultiProcessor Specification. — 1997 — URL: https://pdos.csail.mit.edu/6.828/2018/readings/ia32/MPspec.pdf (accessed: 17.10.2023)