Лабораторная работа 4

ПОЛЬЗОВАТЕЛЬСКИЕ ПРОГРАММЫ. АРГУМЕНТЫ КОМАНДНОЙ СТРОКИ

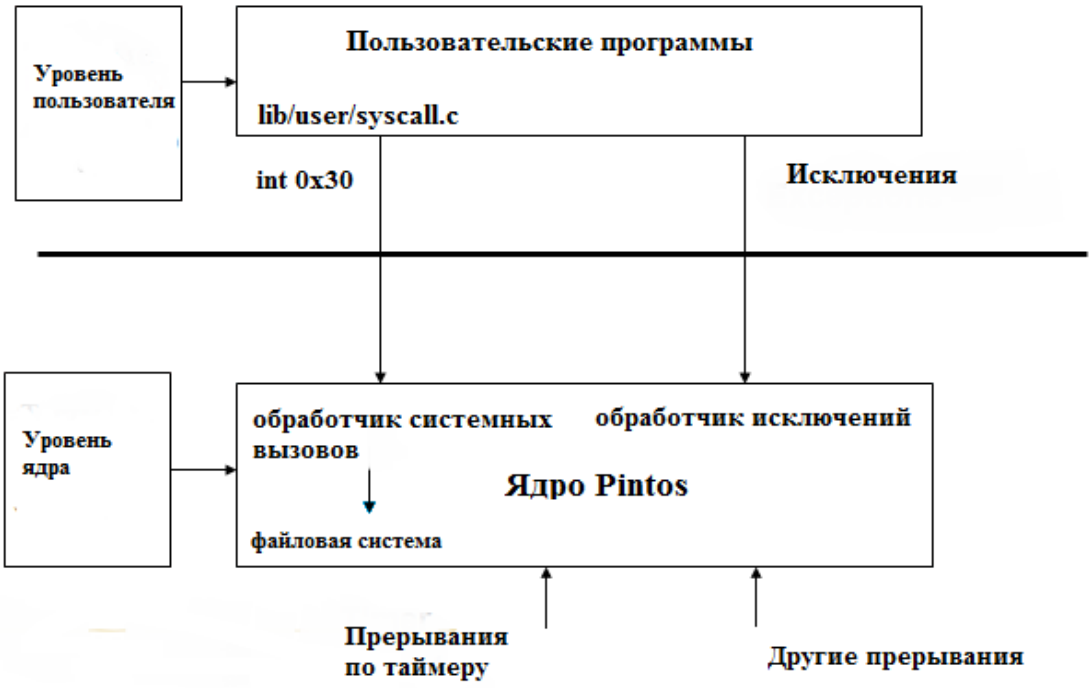
Цель лабораторной работы – изучение механизмов передачи параметров пользовательским программам и реализация такого механизма в архитектуре 80x86 с использованием стека.

Основная рабочая директория: userprog/process.c

Теоретические сведения

Если в предыдущих лабораторных работах пользователь общался с Pintos посредством встроенных тестов, и код, который был написан или модифицирован, принадлежал кодам ядра, то теперь пользователю предлагается расширить возможности Pintos, внедрив в систему возможность обработки пользовательских программ.

Рисунок 4.1. Уровни взаимодействия с ОС Pintos



ОС Pintos может выполнять программы, разработанные на языке программирования C, если они не превышают заданный лимит по размеру и используют ограниченный набор системных вызовов. Например, системный вызов malloc() не может быть использован, так как на данном этапе пользовательские процессы не работают с памятью. Pintos также не может выполнять вычисления над числами с плавающей запятой.

Рабочей директорией данной работы будет не threads, как это было раньше, а userprog. Перед началом работы необходимо подготовить рабочую директорию и скомпилировать набор новых тестов, которые теперь представляют собой процессы пользователя, а не ядра. Для этого необходимо ввести команду make в каталоге userprog.

В каталоге src/examples расположены некоторые примеры пользовательских программ. Используйте команду make, чтобы скомпилировать их, или напишите и скомпилируйте свои собственные примеры. Так как функциональность системы на данном шаге сильно ограничена, не все предоставленные примеры будут выполняться.

Для выполнения данной лабораторной работы необходимо внимательно изучить исходные коды каталога src/userprog, описание которых приведено в табл. 4.1.

Таблица 4.1. Исходные коды каталога src/userprog

Название файла:	**Описание:**
process.c process.h	Функции, которые осуществляют загрузку ELF-файлов (Executable and Linkable Format) и запускают пользовательские процессы
syscall.c syscall.h	Шаблон обработчика системных вызовов, который необходим для взаимодействия пользовательских программ с операционной системой. Базовая реализация обработчика выводит на экран сообщение и завершает пользовательский процесс.
exception.c exception.h	Когда пользовательский процесс вытается выполнить запрещенную или привелегированную операцию, ядро выдает исключение ("fault"). Данные файлы содержат шаблон обработчика подобных исключений, который в базовой реализации выводит сообщение на экран и завершает пользовательский процесс.

Также необходимо использовать файловую систему Pintos. Пользовательские программы не могут быть загружены из хостовой машины, а должны быть скопированы на виртуальный диск с файловой системой. Просмотрите файлы filesys.h и file.h, чтобы получить представление о работе файловой системы и некоторых ограничениях, которые на нее были наложены при разработке:

- Имя файла ограничено 14 символами;
- В файловой системе не реализованы механизмы синхронизации. Разработчику необходимо следить за тем, чтобы в каждый момент времени только один процесс обращался к конкретному объекту файловой системы;
- Размер и количество файлов, которые могут одновременно находиться в файловой системе, ограничено. Это связано тем, что ограничен сам размер файла, а корневой каталог в базовой имплементации представляет собой файл. Система подкаталогов отсутствует;
- Отсутствуют механизмы контроля за внешней фрагментацией;
- Системные ошибки при исполнении пользовательских программ могут повредить диск. Инструментов для восстановления файловой системы не предусмотрено.

Для того чтобы начать работу, необходимо создать диск с файловой системой. Для этой задачи Pintos представляет специальную утилиту pintos-mkdisk. Из директории userprog/build обратитесь к этой утилите с опцией filesys.dsk (имя диска) и --filesys-size=n, которая укажет размер создаваемого диска в Мб:

```
pintos-mkdisk filesys.dsk --filesys-size=2
```

Затем отформатируйте созданный диск, используя опцию –f:

```
pintos --qemu --disk=filesys.dsk -- -f -q
```

При этом в эмуляторе запустится ОС pintos (в качестве диска будет использован виртуальный диск из файла filesys.dsk), ОС pintos отформатирует диск, и завершится.

Попробуйте поместить в файловую систему некоторую пользовательскую программу и запустить ее (примеры пользовательских программ расположены в директории examples):

```
pintos --qemu --disk=filesys.dsk -p ../../examples/echo -a echo -- -q
pintos --qemu --disk=filesys.dsk -- run 'echo x'
```

Если при выполнении была получена ошибка *page fault*, свидетельствующая о том, что пользовательская программа пытается получить доступ к адресному пространству ядра, для данного примера (программы “echo”) повторите повторную загрузку пользовательской программы на диск с файловой системой. Убедитесь, что код пользовательской программы не содержит системные вызовы, которые на данном этапе еще не имплементированы. В случае успешного копирования и запуска ожидается следующий вывод

```
Boot complete.
Executing 'echo x':
Execution of 'echo x' complete.
Timer: 73 ticks
Thread: 0 idle ticks, 73 kernel ticks, 0 user ticks
```

Несмотря на то, что копирование программы в файловую систему и ее выполнение прошло успешно, результат работы программы получен не был. Причиной такого поведения пользовательских программ является то, что в Pintos не имплементированы:

— механизм передачи параметров командной строки запускаемой программы;

— механизм ожидания завершения запущенной программы.

Вам предстоит самостоятельно реализовать средство, которое будет размещать необходимые значения в стек до того, как будет запущен исполняющий программу процесс, а также реализовать код ожидания завершения запущенной программы.

Необходимо еще раз внимательно ознакомиться с функциями ядра, которые представлены в threads/init.c. Из функции main() вызываются следующие функции:

- чтение и синтаксический анализ командной строки (функции read_command_line и parse_options);
- инициализация процессов и консоли ввода-вывода (функции thread_init и console_init);
- инициализация работы с памятью (функции palloc_init, malloc_init и paging_init);
- инициализация глобальной таблицы дескрипторов (функция gdt_init);
- инициализация обработчика прерываний и системного таймера (intr_init и timer_init);
- инициализация исключений и обработчиков системных вызовов (exception_init и syscall_init);
- запуск процесса ядра (функция thread_start);
- инициализация очереди готовых процессов (функция serial_init_queue);
- инициализация клавиатуры и буфера (функции kbd_init и input_init);
- инициализация диск и файловой системы (disk_init и filesys_init);

После завершения инициализации ядро выводит сообщение “Boot complete”. Затем те действия, которые были указаны в командной строке (run, rm, ls, cat, put, get), будут запущены в run_actions. Ядро завершает работу функциями shutdown_power_off.

Рассмотрим подробнее процесс загрузки пользовательских программ в память. Программа загружается в результате выполнения функции run_actions, которая вызывает функцию run_task. Функция run_task в свою очередь вызывает функцию process_execute. Механизм загрузки пользовательских программ в память состоит из нескольких шагов, которые осуществляются над двумя процессами:

- который вызывает функцию process_execute (родитель);
- который загружает и выполняет пользовательскую программу (потомок).

Функция process_execute запускает программу. Функция process_wait необходима ядру для того, чтобы ждать завершения выполнения пользовательской программы. В противном случае ядро может завершить работу несмотря на то, что пользовательская программа все еще выполняется. Такое поведение наблюдается в текущей реализации ОС pintos, и эту проблему необходимо устранить в ходе работы.

Родительский процесс осуществляет следующие действия:

1. Функция process_execute создает новый процесс для пользовательской программы и помещает его в очередь готовых для выполнения. Функция thread_create принимает в качестве аргументов file_name (имя процесса), start_process (функция, которая будет вызвана при старте процесса) и pa (некоторый параметр, который будет передан в start_process).

2. Если создание процесса было успешным, будет создан уникальный номер, который будет идентифицировать процесс пользовательской программы (можно использовать tid созданного дочернего процесса). Такой номер будем называть process_id. Таким образом, process_execute возвращает process_id или -1. Родитель может использовать в дальнейшем process_id как аргумент для функции process_wait.

Процесс-потомок осуществляет следующие действия:

Как только новый процесс (для запуска пользовательской программы) выбран планировщиком, он запускает функцию start_process с аргументами, переданным в параметре pa. Функции start_process потребуется загрузить программу из файловой системы и, если загрузка будет успешной, поместить в стек программы аргументы, поступившие из командной строки. Функция start_process сообщит функции process_execute о своем завершении и передаст значение "success" или "failure".

Для выполнения данной лабораторной работы необходимо тщательно изучить исходные коды этих функций, которые расположены в userprog/process.c. Обратите внимание, что код этих функций нельзя назвать оптимальным: в функции process_wait не реализована защита от завершения работы ядра до старта первого процесса, а в load() — не реализован разбор параметров командной строки и соответствующая настройка стека нового процесса.

Исследуйте работу системы обработки пользовательских программ в режиме отладки и определите те участки, в которые необходимо внести изменения для того, чтобы программы загружались корректно. Особое внимание стоит обратить на механизмы, реализующие связь между функциями: ожидание и сигналы. Составьте диаграмму состояний ожидания, передачи аргументов и результатов между функциями.

Виртуальная память

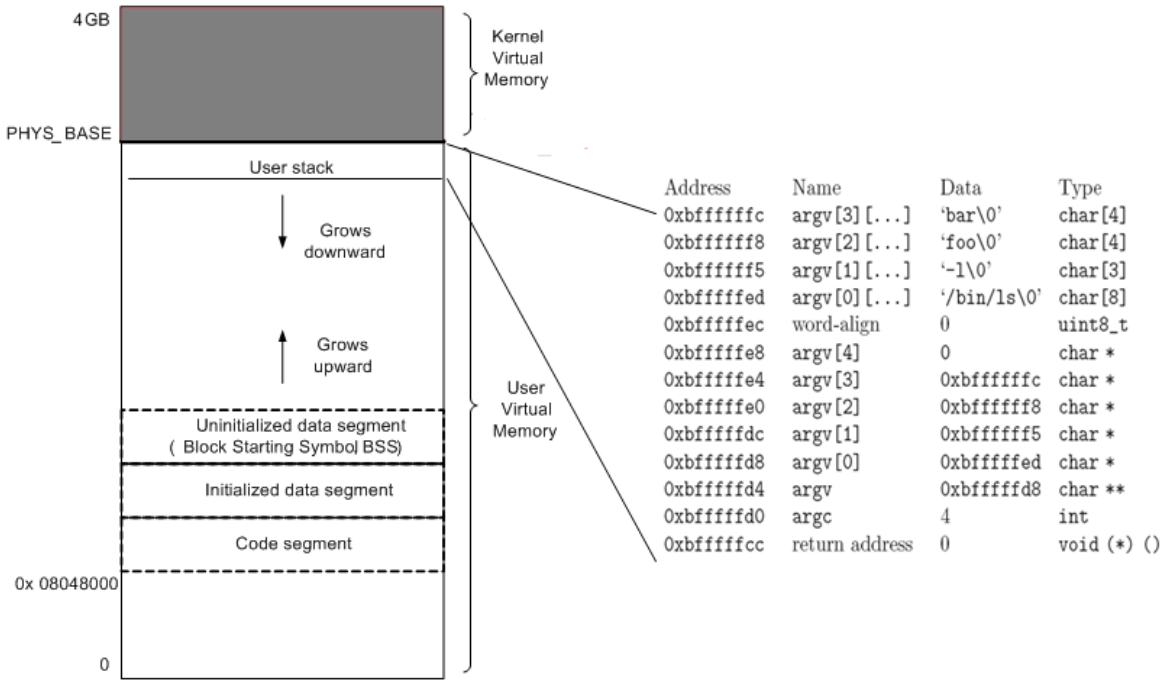
Виртуальная память ОС Pintos разделена на две области: память пользователя и память ядра. Память пользователя располагается в диапазоне от виртуального адреса 0 до адреса PHYS_BASE, память ядра занимает оставшееся адресное пространство от PHYS_BASE до 4 Гб.

Виртуальная память пользователя имеет локальный характер. Когда ядро выполняет переключение контекста процессов, оно также переключается с одного виртуального адреса на другой (функция pagedir_activate() в userprog/pagedir.c). Виртуальная память ядра глобальна и отображается аналогично памяти программы пользователя, начиная с адреса PHYS_BASE. Виртуальный адрес PHYS_BASE обращается к физическому адресу 0, виртуальный адрес PHYS_BASE+0x1234 обращается к физическому 0x1234 и т.д.

Пользовательская программа имеет доступ только к ее собственной виртуальной памяти. Попытка получить доступ к памяти ядра ведет к страничному исключению, которое обрабатывается в функции page_fault(), расположенной в userprog/exception.c. Процессы ядра имеют доступ к виртуальной памяти ядра и к виртуальной памяти процесса, который исполняется в данный момент.

При работе с системными вызовами ядру часто приходится получать доступ к памяти через указатели, предоставляемые программой пользователя. Эти операции ядро должно выполнять с особой осторожностью, так как пользовательская программа может передавать нулевой указатель или указатель на адресное пространство ядра (над PHYS_BASE). На рис. 4.2. представлена схема адресного пространства пользовательского процесса. Доступ по недействительным указателям должен быть запрещен без вреда ядру или другим исполняющимся в этот момент процессам путем завершения процесса-нарушителя и освобождения его ресурсов.

Рисунок 4.2. Виртуальное адресное пространство



Один из возможных методов реализации такой защиты – проверка действительности предоставляемого пользовательским процессом указателя и последующее разыменование его. Это наиболее простой метод обработки доступа к пользовательской памяти, который потребует подробного знакомства с функциями, представленными в userprog/pagedir.c и threads/vaddr.h.

Работа с командной строкой и стеком процесса в ОС Pintos

Операционная система Pintos осуществляет разбор аргументов командной строки в функции main, расположенной в файле "threads/init.c". В базовой реализации Pintos используется функция parse_options, которая обрабатывает все опции командной строки ОС Puntos, такие как –h или -f.

```
static char **
parse_options (char **argv)
{
  for (; *argv != NULL && **argv == '-'; argv++)
    {
      char *save_ptr;
      char *name = strtok_r (*argv, "=", &save_ptr);
      char *value = strtok_r (NULL, "", &save_ptr);

      if (!strcmp (name, "-h"))
        usage ();
      else if (!strcmp (name, "-q"))
        power_off_when_done = true;
#ifdef FILESYS
      else if (!strcmp (name, "-f"))
        format_filesys = true;
#endif
      else if (!strcmp (name, "-rs"))
        random_init (atoi (value));
      else if (!strcmp (name, "-mlfqs"))
        thread_mlfqs = true;
#ifdef USERPROG
      else if (!strcmp (name, "-ul"))
        user_page_limit = atoi (value);
#endif
      else
        PANIC ("unknown option `%s' (use -h for help)", name);
    }

  return argv;
}
```

Затем в функции main управление передается функции run_actions с аргументом argv в виде строки, которая состоит из имени пользовательской программы и ее аргументов. Например, “run echo x“. В функции run_actions осуществляется разбиение переданной по указателю строки на задание: собственное имя задачи или имя пользовательской программы, а также ее аргументы. В данном примере “run” – это имя задания, ”echo” – имя пользовательской программы, а ”x” – аргумент пользовательской программы.

```
static void run_actions (char **argv)
{
```

```
  struct action
    {
      char *name;                       /*имя задания */
      int argc;                         /*кол-во аргументов*/
      void (*function) (char **argv);   /*функция*/
    };


/*Таблица поддерживаемых заданий*/
  static const struct action actions[] =
    {
      {"run", 2, run_task},
#ifdef FILESYS
      {"ls", 1, fsutil_ls},
      {"cat", 2, fsutil_cat},
      {"rm", 2, fsutil_rm},
      {"put", 2, fsutil_put},
      {"get", 2, fsutil_get},
#endif
      {NULL, 0, NULL},
    };

  while (*argv != NULL)
    {
      const struct action *a;
      int i;

      /* Поиск имени задания*/
      for (a = actions; ; a++)
        if (a->name == NULL)
          PANIC ("unknown action `%s' (use -h for help)", *argv);
        else if (!strcmp (*argv, a->name))
          break;

      /*Проверка требуемых аргументов*/
      for (i = 1; i < a->argc; i++)
        if (argv[i] == NULL)
          PANIC ("action `%s' requires %d argument(s)", *argv, a->argc - 1);

      /* Запуск задания */
      a->function (argv);
      argv += a->argc;
    }

}
```

Что очевидно из кода функции, указание задание “run” в командной строке, т.е. запрос на выполнение программы, вызывает процедуру run_task, куда с помощью указателя argv передается строка “echo x”. Процедура run_task в свою очередь вызывает process_execute, которая расположена в “userprog/process.c”, и передает ей указатель строку с аргументами.

Процедура process_execute() создает новый процесс с помощью функции start_process, которая выполняет загрузку пользовательской программы. Именно на стадии загрузки программы необходимо выполнить ряд изменений, чтобы осуществить передачу параметров.

Во-первых, необходимо извлечь аргумент file_name из передаваемой строки (т. е. в данном примере это аргумент “echo” из строки “echo x”) для того, чтобы выполнять корректную загрузку пользовательской программы в функции load. Если не отделить первое слово из командной строки, то на диске будет произведен поиск файла с именем “echo x”, а не “echo”. Для отделения первого слова можно использовать функции, представленные в “lib/string.c”, но можно реализовать этот функционал локально с помощью циклов и условий.

После этого необходимо инициализировать работу со стеком. Начальная инициализация осуществляется при вызове функции setup_stack. Функция выполняет создание и настройку стека программы и возвращает значение регистра esp. При работе программ регистр esp всегда содержит указатель на текущую вершину стека. В общем случае, стек располагается на самой вершине виртуального адресного пространства пользователя, на странице, расположенной прямо под адресом PHYS_BASE (см. “threads/vaddr.h”).

```
...
  /* Создание и настройка стека. */
  if (!setup_stack (esp))
    goto done;

  /* Установка регистра eip – адреса начала кода для выполнения. */
  *eip = (void (*) (void)) ehdr.e_entry;

  success = true;

  // Здесь должна быть реализована настройка стека:
  // записаны параметры командной строки и их количество
...
```

Собственные изменения можно вносить только после того, как флаг успешной инициализации (success) будет установлен в единицу.

Теперь требуется поместить аргументы в стек в правильном порядке (рис. 4.2). Стек заполняется в обратном порядке, от наибольших адресов памяти в сторону уменьшения адресов, «снизу вверх»

Вначале необходимо поместить в стек мнимый адрес возврата (return address). Адрес возврата в данной задаче будет мнимым, потому что, несмотря на то, что данная функция не возвращает значение, и обращение к аргументам происходит через указатели, ее стек должен иметь такую же структуру, как стек любой другой функции. Поэтому наличие адреса возврата необходимо, но для этой, первой функции — точки входа, никогда не используется, т.е. может быть передано любое значение (в примере на рис 4.2 записан 0).

Затем в стек помещается целочисленное значение argc — количество аргументов командной строки запускаемой программы. Имя самой программы всегда является первым аргументом. Например, если программа запущена без аргументов как "echo", то argc = 1, если как "echo x", то argc = 2 и т.д.

Далее в стек помещается указатель на начало массива argv — массива Си-строк, каждый элемент которого содержит переданный при запуске программы аргумент. Количество элементов массива должно быть равно argc+1. Последний элемент массива содержит нулевой указатель.

Затем в стеке размещаются данные самих строк. Порядок их размещения в стеке не так важен, т.к. массив указателей argv содержит адреса этих строк в правильном порядке, обращаться к аргументам

программа будет только с помощью argv. Не стоит забывать о символах конца строки (нулевой байт окончания строки).

Любая программа будет завершаться с ошибкой "page fault" пока не будет корректно заполнен стек запускаемой программы.

В целях отладки можно после строки в process_start:

```
success = true;
```

указать:

```
*esp -= 12;
```

Такое изменение esp означает, что в стек программы записаны три 4х-байтовых значения. Т.к. страница памяти при выделении была обнулена, то эти значения содержат нули, т.е. argc=0, argv=0. Такое временное отладочное решение позволит запускать пользовательские программы в pintos, но без аргументов командной строки. Например, запуск программы echo без аргументов следующей строкой запуска будет успешен:

```
pintos --qemu --disk=filesys.dsk -- run 'echo'
```

Pintos предоставляет удобное средство отладки – функцию hex_dump(), которая определена в <stdio.h> и позволяет вывести на экран содержимое по указанному адресу памяти. Например, стек текущего процесса может быть распечатан следующим вызовом:

```
hex_dump( 0, *esp, PHYS_BASE - *esp, false );
```

Ожидание завершения пользовательских программ

После запуска программы функция run_task переходит к ожиданию завершения запущенной программы с помощью process_wait. Эта функция должна дождаться завершения указанной по process_id программы и вернуть код выхода (exit status). Кодом выхода называется значение, возвращенное из функции main запущенной пользовательской программы. Функция process_wait возвращает -1, если входным параметром указан несуществующий процесс.

В настоящий момент реализация функции отсутствует:

```
int process_wait (tid_t child_tid UNUSED)
{
  return -1;
}
```

В результате, если запустить ядро pintos с параметром ядра -q (выключить компьютер после завершения работы все процессов), то виртуальный компьютер будет выключен сразу после запуска программы, не дожидаясь ее завершения:

```
pintos --qemu --disk=filesys.dsk -- -q run 'echo'
```

Вам предлагается самостоятельно реализовать функцию ожидания запущенной программы, используя изученные в предыдущих работах механизмы синхронизации. После реализации функции ожидания указанный выше вызов pintos будет приводить к корректным результатам: ОС pintos будет выключать эмулятор только после завершения пользовательской программы.

Системные вызовы

Системные вызовы обеспечивают взаимосвязь между ядром и программами пользователя**.** Их можно рассматривать как функции, которые вызываются из пользовательских программ и выполняются ядром. Данная работа не ставит целью реализовать поддержку всех системных вызовов в ОС pintos, однако, для проверки корректности реализации разбора аргументов командной строки, необходима возможность иметь минимальную поддержку для системных вызовов: программа пользователя должна иметь возможность печатать диагностические сообщения и корректно завершаться. Для этого в ядре системы должны быть реализованы обработчики для двух системных вызовов:

exit — вызывается пользовательской программой при завершении. Вызов содержит единственный аргумент — код выхода (значение, возвращенное функцией main программы).

write — вызывается пользовательской программой внутри вызова printf() и необходима для вывода данных в консоль. Содержит 3 аргумента: дескриптор (обычно 1), адрес буфера для печати и количество байтов в буфере.

Для реализации обработчиков следует обратиться к файлу userprog/syscall.c. Функция syscall_handler, обрабатывающая запрос пользовательской программы, реализована следующим образом:

```
static void
syscall_handler (struct intr_frame *f UNUSED)
{
  printf ("system call!\n");
  thread_exit ();
}
```

При текущей реализации любой вызов от пользовательской программы к ядру приведет к печати сообщения "system call" в консоль и немедленному завершению программы.

Для поддержки системного вызова write следует добавить в начало тела функции его минимальную реализацию:

```
if ( *(int*) f->esp == SYS_WRITE) {
   putbuf( ((const char**) f->esp)[2], ((size_t*) f->esp)[3]);
   return;
}
```

Реализацию системного вызова exit следует выполнить самостоятельно. Реализация должна предусматривать сохранение в каких-либо структурах данных кода выхода программы пользователя (с целью дальнейшего возврата этого кода из функции process_wait):

```
if (*(int*) f->esp == SYS_EXIT) {
   int exit_status = ((size_t*) f->esp)[1];
   ...
   thread_exit();// вызов exit() должен завершать программу
}
```

Обработчики других системных вызовов потребуется реализовать в следующей работе. В данной же работе для любого другого системного вызова может использоваться реализация по умолчанию: вывод "system call! " и принудительное завершение программы.

Порядок выполнения работы

1. В функцию process_exit (userprog/process.c) необходимо добавить печать диагностического сообщения о завершении пользовательской программы:
printf("%s: exit(%d)\n", ..., …), где %s — имя пользовательской программы, %d — код выхода, возвращенный завершенной функцией main пользовательской программы.
Формат вывода должен быть в точности таким — это необходимо для успешного прохождения тестов. Имя пользовательской программы не должно содержать аргументов командной строки.
Например, при завершении программы, запущенной как "echo x" после ее завершения должно выводиться: "echo: exit(0)".
Сообщения должны выводиться только для пользовательских процессов.
Для хранения имен запущенных программ и их кодов завершения можно использовать какие-либо структуры данных, разработанные вами.

2. Модифицировать работу системы обработки команд пользователя и передачи параметров командной строки так, чтобы пользовательская программа могла считывать переданные ей аргументы. Пользовательские программы в Pintos вызываются следующим образом:

```
pintos [опции] -- run '[имя программы] [аргументы]'
```

Например:

```
pintos --qemu --disk=filesys.dsk -- -q run 'ls -l foo bar'
```

Исходную командную строку необходимо разбить на слова "ls", "-l", "foo", "bar" в функции load. Проанализировать элементы командной строки (например, используя функцию strtok_r() в lib/string.c) и сохранить в массиве, и только потом помещать их в стек, осуществляя копирование из массива.
Аргументы помещаются в стек один за другим в порядке справа налево. После добавления каждого аргумента уменьшать значение указателя вершины стека, затем сохранять аргумент по адресу, на который он указывает в данный момент.

3. Реализовать функцию ожидания завершения пользовательских программ.

4. Реализовать минимальную поддержку системных вызов (write, exit) от пользовательских программ к ядру ОС pintos. Функция exit должна сохранять код выхода завершившейся пользовательской программы для его дальнейшего вывода (п. 1) и возврата кода выхода в process_wait.

5. Убедиться в том, что операционная система верно воспринимает аргументы командной строки, используя следующие тесты:

- args-none
- args-single
- args-multiple
- args-many
- args-dbl-space

Во избежание возникновения ошибок при запуске тестов, в командной строке должен быть явно указан эмулятор:

```
make tests/userprog/[test_name].result SIMULATOR=--qemu
```

6. Реализовать дополнительное индивидуальное задание в указанном файле исходного кода. Описать в отчете алгоритм программы, привести результаты работы и проанализировать их.

Содержание отчета

В отчете необходимо указать следующие данные:

1. Исходные коды ОС Pintosc внесенными модификациями и комментариями.
2. Диаграмма состояний ожидания, передачи аргументов и результатов выполнения между основными функциями (thread_create, run_task, process_execute, process_wait, start_process. thread_exit).
3. Анализ тестовых программ: какие командные строки подают тестовые программы и как реализована обработка этих случаев.
4. Ответы на контрольные вопросы.
5. Результаты выполнения дополнительного индивидуального задания.