1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. **Высшая школа кибербезопасности и защиты информации**

**ЛАБОРАТОРНАЯ РАБОТА № 4**

1. «**Пользовательские программы. Аргументы командной строки**»
2. по дисциплине «Операционные системы»
3. Выполнил
4. студент гр. 5131001/20001 Маронова К.Д.

<*подпись*>

1. Преподаватель
2. Огнёв Р.А.

<*подпись*>

1. Санкт-Петербург
2. 2023

**1. Цель работы**

Изучение механизмов передачи параметров пользовательским программам и реализация такого механизма в архитектуре 80x86 с использованием стека.

**2. Постановка задачи**

Основная задача лабораторной работы состоит из нескольких пунктов:

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.

2. Модифицировать работу системы обработки команд пользователя и передачи параметров командной строки так, чтобы пользовательская программа могла считывать переданные ей аргументы.

Исходную командную строку необходимо разбить на слова в функции load. Проанализировать элементы командной строки, сохранить их 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 и возврата кода выхода в process\_wait.

**3. Описание решения**

Для задачи №1 были внесены изменения в функцию process\_exit. Для корректной работы printf произведено разбиение строки, которая лежит в name текущего потока, по пробелу. Для вывода кода выхода в структуру thread было добавлено поле int code\_for\_exiting, которое заполняется в syscall.c.

В задаче №2 был реализован стэк для работы с аргументами командной строки. Для начала, необходимо было разделить название потока по пробелу, для того чтобы вычленить имя программы (первый аргумент) и передать его в функцию filesys\_open. Затем был реализован сам стэк исходя из требований, предоставленных в методических указаниях:

Сначала происходит разделение исходной строки file\_name на отдельные слова с помощью функции strtok\_r, разделителем является пробел. Полученные слова записываются в массив argv, а количество аргументов сохраняется в переменной argc.

Затем происходит запись аргументов в стек процесса. Сначала вычисляется длина каждого слова, включая завершающий нулевой символ, и уменьшается указатель на вершину стека (\*esp) на эту длину, чтобы освободить место для записи аргумента. Аргумент копируется в стек с помощью функции strlcpy. При этом для каждого аргумента сохраняется указатель на строку в стеке (\*esp) в массиве array.

Затем записываются адреса строк в стек. Указатель set\_pointer указывает на место в стеке, где будут записаны адреса. При этом указатель set\_pointer сначала уменьшается на размер указателя (sizeof(char\*)), а затем с помощью функции memcpy копируются адреса строк в стек.

Далее записывается указатель ptr на массив указателей array в стек, а затем записывается количество аргументов argc в стек. Наконец, указатель set\_pointer становится адресом возврата, и он присваивается указателю \*esp, указывающему на начало стека процесса.

Для реализации задачи №3 используется семафор. Он инициализируется в функции process\_execute. В функции wait производим операцию над семафором sema\_down для того, чтобы гарантировать, что никакой следующий процесс не получит доступ к семафору, пока текущий владелец его не освободит (пока программа не завершится), затем в exit мы «поднимаем» семафор.

Реализация задачи №4 взята из методического пособия.

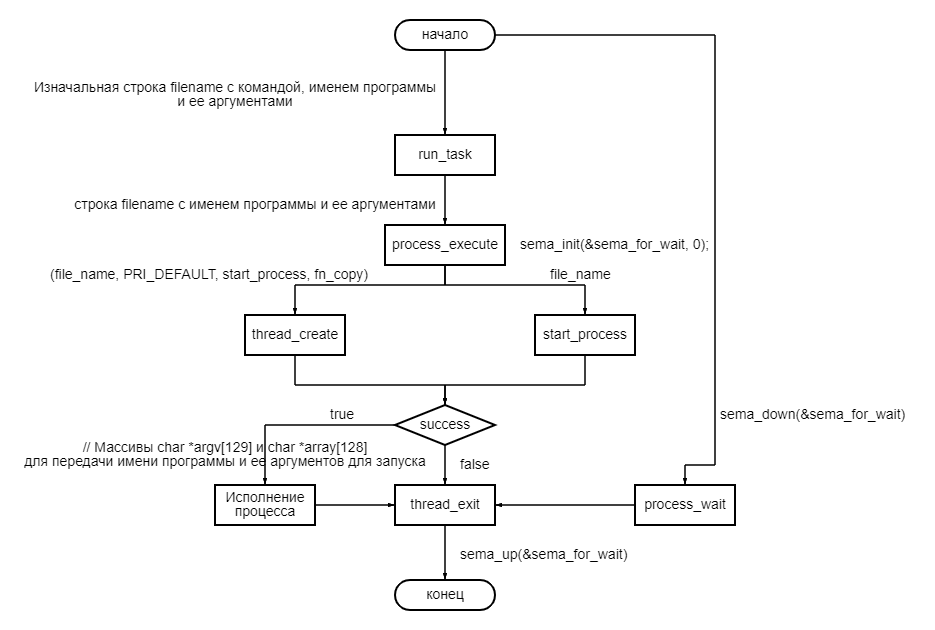


Рис. 1. «Диаграмма состояний ожидания, передачи аргументов и результатов выполнения между основными функциями».

**4. Тестирование и результаты работы программы**

Все необходимые тесты были успешно пройдены:

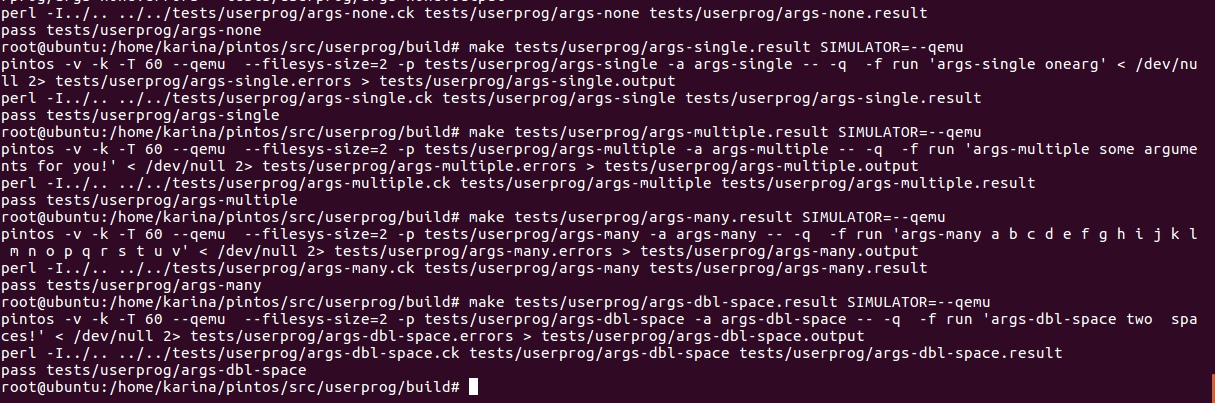


Рис. 2. «Результаты тестов».

**5. Выводы**

В ходе лабораторной работы удалось изучить механизмы передачи параметров пользовательским программам и реализовать такой механизм в архитектуре 80x86 с использованием стека.

Основные проблемы, встретившиеся в ходе работы заключались в следующем: 1) Не сразу удалось задать верное направление аргументов в стэке. 2)Проблема с передачей указателей на массивы и код выхода в стэк (была решена дополнительным указателем). 3) Отсутствие в некоторых необходимых местах завершающего нуля (была решена посредством использования функции strlcpy, которая копирует строки вместе с завершающим нулем).

Листинг программы:

tid\_t

process\_execute (const char \*file\_name)

{

char \*fn\_copy;

tid\_t tid;

struct thread\* cur = thread\_current();

sema\_init(&sema\_for\_wait, 0); //инициализация семафора

/\*

поток будет блокирован до тех пор, пока семафор `sema\_for\_wait` не станет

положительным (другими словами, пока другой процесс не вызовет `sema\_up(&sema\_for\_wait)`).

Использование семафора `sema\_for\_wait` позволяет дождаться выполнения пользовательской программы.

Функция `process\_exit` вызывает `sema\_up(&sema\_for\_wait)`, чтобы разблокировать поток, который ждет

завершения текущего процесса. Это происходит после освобождения ресурсов текущего процесса.

\*/

/\* Make a copy of FILE\_NAME.

В противном случае возникает гонка данных между вызывающим и функцией load().. \*/

fn\_copy = palloc\_get\_page (0);

if (fn\_copy == NULL)

return TID\_ERROR;

strlcpy (fn\_copy, file\_name, PGSIZE);

/\* Create a new thread to execute FILE\_NAME. \*/

tid = thread\_create (file\_name, PRI\_DEFAULT, start\_process, fn\_copy);

if (tid == TID\_ERROR)

palloc\_free\_page (fn\_copy);

return tid;

}

int

process\_wait (tid\_t child\_tid)

{

if (child\_tid != -1) {

//struct thread\* cur = get\_thread\_by\_tid(child\_tid);

sema\_down(&sema\_for\_wait); //необходим для того, чтобы дождаться выполнения пользовательской программы

return;

}

else return -1;

}

/\* Free the current process's resources. \*/

void

process\_exit (void)

{

struct thread \*cur = thread\_current ();

uint32\_t \*pd;

/\* Destroy the current process's page directory and switch back

to the kernel-only page directory. \*/

pd = cur->pagedir;

if (pd != NULL)

{

/\* Correct ordering here is crucial. We must set

cur->pagedir to NULL before switching page directories,

so that a timer interrupt can't switch back to the

process page directory. We must activate the base page

directory before destroying the process's page

directory, or our active page directory will be one

that's been freed (and cleared). \*/

cur->pagedir = NULL;

pagedir\_activate (NULL);

pagedir\_destroy (pd);

//////////NEW//////////////////

char\* token, \* save\_ptr, \* name\_copy;

name\_copy = palloc\_get\_page(0);

if (name\_copy == NULL)

return TID\_ERROR;

strlcpy(name\_copy, cur->name, PGSIZE);

// получение имени программы

token = strtok\_r(name\_copy, " ", &save\_ptr);

if (token != NULL) {

printf("%s: exit(%d)\n", token, cur->code\_for\_exiting);

}

/\*printf("%s: exit(%d)\n", cur->name, cur->code\_for\_exiting);\*/

sema\_up(&sema\_for\_wait);

///////////////////////////////

}

}

bool

load (const char \*file\_name, void (\*\*eip) (void), void \*\*esp)

{

struct thread \*t = thread\_current ();

struct Elf32\_Ehdr ehdr;

struct file \*file = NULL;

off\_t file\_ofs;

bool success = false;

int i;

/\* Allocate and activate page directory. \*/

t->pagedir = pagedir\_create ();

if (t->pagedir == NULL)

goto done;

process\_activate ();

//////////////////NEW/////////////////////

char\* s\_ptr;

char\* tk;

char n\_copy[128];

//n\_copy = palloc\_get\_page(0);

if (n\_copy == NULL)

return TID\_ERROR;

strlcpy(n\_copy, file\_name, PGSIZE);

// получение имени программы

tk = strtok\_r(n\_copy, " ", &s\_ptr);

file = filesys\_open(tk);

/////////////////////////////////////////

/\* Open executable file. \*/

/\*file = filesys\_open (file\_name);\*/

if (file == NULL)

{

printf ("load: %s: open failed\n", tk);

goto done;

}

/\* Read and verify executable header. \*/

if (file\_read (file, &ehdr, sizeof ehdr) != sizeof ehdr

|| memcmp (ehdr.e\_ident, "\177ELF\1\1\1", 7)

|| ehdr.e\_type != 2

|| ehdr.e\_machine != 3

|| ehdr.e\_version != 1

|| ehdr.e\_phentsize != sizeof (struct Elf32\_Phdr)

|| ehdr.e\_phnum > 1024)

{

printf ("load: %s: error loading executable\n", tk);

goto done;

}

/\* Read program headers. \*/

file\_ofs = ehdr.e\_phoff;

for (i = 0; i < ehdr.e\_phnum; i++)

{

struct Elf32\_Phdr phdr;

if (file\_ofs < 0 || file\_ofs > file\_length (file))

goto done;

file\_seek (file, file\_ofs);

if (file\_read (file, &phdr, sizeof phdr) != sizeof phdr)

goto done;

file\_ofs += sizeof phdr;

switch (phdr.p\_type)

{

case PT\_NULL:

case PT\_NOTE:

case PT\_PHDR:

case PT\_STACK:

default:

/\* Ignore this segment. \*/

break;

case PT\_DYNAMIC:

case PT\_INTERP:

case PT\_SHLIB:

goto done;

case PT\_LOAD:

if (validate\_segment (&phdr, file))

{

bool writable = (phdr.p\_flags & PF\_W) != 0;

uint32\_t file\_page = phdr.p\_offset & ~PGMASK;

uint32\_t mem\_page = phdr.p\_vaddr & ~PGMASK;

uint32\_t page\_offset = phdr.p\_vaddr & PGMASK;

uint32\_t read\_bytes, zero\_bytes;

if (phdr.p\_filesz > 0)

{

/\* Normal segment.

Read initial part from disk and zero the rest. \*/

read\_bytes = page\_offset + phdr.p\_filesz;

zero\_bytes = (ROUND\_UP (page\_offset + phdr.p\_memsz, PGSIZE)

- read\_bytes);

}

else

{

/\* Entirely zero.

Don't read anything from disk. \*/

read\_bytes = 0;

zero\_bytes = ROUND\_UP (page\_offset + phdr.p\_memsz, PGSIZE);

}

if (!load\_segment (file, file\_page, (void \*) mem\_page,

read\_bytes, zero\_bytes, writable))

goto done;

}

else

goto done;

break;

}

}

/\* Set up stack. \*/

if (!setup\_stack (esp))

goto done;

/\* Start address. \*/

\*eip = (void (\*) (void)) ehdr.e\_entry;

success = true;

//////////////NEW//////////////////

int argc = 0; //кол-во аргументов

char \*argv[129];//массив для аргументов сюда мы записываем сами строки

char\* token\_for\_stack, \* save\_ptr;//переменные, необходимые для функции strtok

char \*array[128]; //массив для адресов строк из argv в стэке

//Вспомогательная строка, т.к. исходную не стоит менять

char\* tmp\_filename = palloc\_get\_page(sizeof(char) \* (strlen(file\_name) + 1));

strlcpy(tmp\_filename, file\_name, (strlen(file\_name) + 1));

for (token\_for\_stack = strtok\_r(tmp\_filename, " ", &save\_ptr); token\_for\_stack != NULL; token\_for\_stack = strtok\_r(NULL, " ", &save\_ptr)) {

//цикл по аргументам для их подсчёта и записи в массив строк

argv[argc] = token\_for\_stack;

argc++;

}

int len; //длина слова

char\* set\_pointer; //указатель на адрес памяти, который используется для установки значений в стеке процесса. Это как бы временная копия стэка

void\* pointer; //Это указатель на указатель(массив указателей), который содержит адреса строк в стеке.Он используется для сохранения адресов строк в обратном порядке.

//запись в есп наших аргументов в обратном порядке

/\*

указатель на вершину стека `\*esp` уменьшается на `len`, чтобы освободить место

для записи аргумента. Аргумент копируется в стек с помощью функции `strlcpy`.

В массиве `array` сохраняется указатель на строку в стеке `\*esp` для каждого

аргумента.

\*/

for (int i = argc - 1; i >= 0; i--) {

len = strlen(argv[i]) + 1;

\*esp -= len;

strlcpy(\*esp, argv[i], len);

array[i] = \*esp; //тут происходит запись в массив указателей argv указатель на нашу строчку в стэке

}

set\_pointer = \*esp; //запоминаем последнее положение esp

set\_pointer -= sizeof(char\*);

//Записываем адреса строк в стек

for (int i = argc - 1; i >= 0; i--) {

set\_pointer -= sizeof(char\*);

memcpy(set\_pointer, &array[i], sizeof(char\*));

}

void\*\* ptr = set\_pointer;

set\_pointer -= sizeof(char\*);

memcpy(set\_pointer, &ptr, sizeof(char\*)); //значение `ptr` копируется в память, на которую указывает `set\_pointer`

set\_pointer -= sizeof(char\*);

\*set\_pointer = argc;//Запись количества аргументов

set\_pointer -= sizeof(char\*);

\*esp = set\_pointer;//адрес возврата. Указатель на начало стэка процесса

///////////////////////////////////////////////

done:

/\* We arrive here whether the load is successful or not. \*/

file\_close (file);

return success;

}

***SYSCALL.C:***

static void

syscall\_handler (struct intr\_frame \*f UNUSED)

{

/\*printf ("system call!\n");

thread\_exit ();\*/

if (\*(int\*)f->esp == SYS\_WRITE) {

putbuf(((const char\*\*)f->esp)[2], ((size\_t\*)f->esp)[3]);

return;

}

else if (\*(int\*)f->esp == SYS\_EXIT) {

int exit\_status = ((size\_t\*)f->esp)[1];

thread\_current()->code\_for\_exiting = exit\_status;

thread\_exit();

}

else return;

}