Министерство образования Республики Беларусь

ПОЛОЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра технологий программирования

Методические указания для выполнения лабораторной работы № 4 по курсу «Операционные системы и системное программирование»

«Процессы в ОС UNIX: управление процессами»

Полоцк, 2019

Процессы

Контекст процесса

Контекст процесса складывается из пользовательского контекста и контекста ядра, как изображено на рисунке.

Под пользовательским контекстом процесса понимают код и данные, расположенные в адресном пространстве процесса. Все данные подразделяются на:

инициализируемые неизменяемые данные (например, константы);

инициализируемые изменяемые данные (все переменные, начальные значения которых присваиваются на этапе компиляции);

неинициализируемые изменяемые данные (все статические переменные, которым не присвоены начальные значения на этапе компиляции);

стек пользователя;

данные, расположенные в динамически выделяемой памяти (например, с помощью стандартных библиотечных C функций malloc(), calloc(), realloc()).

Исполняемый код и инициализируемые данные составляют содержимое файла программы, который исполняется в контексте процесса. Пользовательский стек применяется при работе процесса в пользовательском режиме (user-mode).

Под понятием "контекст ядра" объединяются системный контекст и регистровый контекст, рассмотренные на лекции. Мы будем выделять в контексте ядра стек ядра, который используется при работе процесса в режиме ядра (kernel mode), и данные ядра, хранящиеся структурах, являющихся аналогом блока управления процессом — PCB. Состав данных ядра будет уточняться на последующих семинарах. На этом занятии нам достаточно знать, что в данные ядра входят: идентификатор пользователя — UID, групповой идентификатор пользователя — GID, идентификатор процесса — PID, идентификатор родительского процесса — PPID.

Идентификация процесса

Каждый процесс в операционной системе получает уникальный идентификационный номер – PID (process identificator). При создании нового процесса операционная система пытается присвоить ему свободный номер больший, чем у процесса, созданного перед ним. Если таких свободных номеров не оказывается (например, мы достигли максимально возможного номера для процесса), то операционная система выбирает минимальный номер из всех свободных номеров. В операционной системе Linux присвоение идентификационных номеров процессов начинается с номера 0, который получает процесс kernel при старте операционной системы. Этот номер впоследствии не можетбыть присвоен никакому другому процессу. Максимально возможное значение для номера процесса в Linux на базе 32-разрядных процессоров Intel составляет 231-1.

Иерархия процессов

операционной системе UNIX все процессы, кроме одного, создающегося при старте операционной системы, могут быть порождены только какими-либо другими процессами. В качестве прародителя всех остальных процессов в подобных UNIX

системах могут выступать процессы с номерами 1 или 0. В операционной системе Linux таким родоначальником, существующим только при загрузке системы, является процесс kernel с идентификатором 0.

Таким образом, все процессы в UNIX связаны отношениями процесс-родитель – процесс-ребенок и образуют генеалогическое дерево процессов. Для сохранения целостности генеалогического дерева в ситуациях, когда процесс-родитель завершает свою работу до завершения выполнения процесса-ребенка, идентификатор родительского процесса в данных ядра процесса-ребенка (PPID – parent process identificator) изменяет свое значение на значение 1, соответствующее идентификатору процесса init, время жизни которого определяет время функционирования операционной системы. Тем самым процесс init как бы усыновляет осиротевшие процессы. Наверное, логичнее было бы заменять PPID не на значение 1, а на значение идентификатора ближайшего существующего процесса-прародителя умершего процесса-родителя, но в UNIX почему-то такая схема реализована не была.

Системные вызовы getppid() и getpid()

Данные ядра, находящиеся в контексте ядра процесса, не могут быть прочитаны процессом непосредственно. Для получения информации о них процесс должен совершить соответствующий системный вызов. Значение идентификатора текущего процесса может быть получено с помощью системного вызова getpid(), а значение идентификатора родительского процесса для текущего процесса – с помощью системного вызова getppid(). Прототипы этих системных вызовов и соответствующие типы данных описаны в системных файлах <sys/types.h> и <unistd.h>. Системные вызовы не имеют параметров и возвращают идентификатор текущего процесса и идентификатор родительского процесса соответственно.

Системные вызовы getpid() и getppid()
Прототипы системных вызовов

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

Описание системных вызовов

Системный вызов getpid возвращает идентификатор текущего процесса. Системный вызов getppid возвращает идентификатор процесса-родителя для текущего процесса. Тип данных pid_t является синонимом для одного из целочисленных типов языка C.

Создание процесса в UNIX. Системный вызов fork()

операционной системе UNIX новый процесс может быть порожден единственным способом – с помощью системного вызова fork(). При этом вновь созданный процесс будет являться практически полной копией родительского процесса.У порожденного процесса по сравнению с родительским процессом (на уровне уже полученных знаний) изменяются значения следующих параметров: идентификатор процесса – PID; идентификатор родительского процесса – PPID.

Системный вызов для порождения нового процесса Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Описание системного вызова

Системный вызов fork служит для создания нового процесса в операционной системе UNIX. Процесс, который инициировал системный вызов fork, принято называть родительским процессом (parent process). Вновь порожденный процесс принято называть процессом-ребенком (child process). Процесс-ребенок является почти полной копией родительского процесса. У порожденного процесса по сравнению с родительским изменяются значения следующих параметров:

идентификатор процесса;
идентификатор родительского процесса;
время, оставшееся до получения сигнала SIGALRM;
сигналы, ожидавшие доставки родительскому процессу, не будут доставляться порожденному процессу.

При однократном системном вызове возврат из него может произойти дважды: один раз в родительском процессе, а второй раз в порожденном процессе. Если создание нового процесса произошло успешно, то в порожденном процессе системный вызов вернет значение 0, а в родительском процессе – положительное значение, равное идентификатору процесса-ребенка. Если создать новый процесс не удалось, то системный вызов вернет в инициировавший его процесс отрицательное значение.

Системный вызов fork является единственным способом породить новый процесс после инициализации операционной системы UNIX в процессе выполнения системного вызова fork() порождается копия родительского процесса и возвращение из системного вызова будет происходить уже как в родительском, так и в порожденном процессах. Этот системный вызов является единственным, который вызывается один раз, а при успешной работе возвращается два раза (один раз в

процессе-родителе и один раз в процессе-ребенке)! После выхода из системного вызова оба процесса продолжают выполнение регулярного пользовательского кода, следующего за системным вызовом.

Для того чтобы после возвращения из системного вызова fork() процессы могли определить, кто из них является ребенком, а кто родителем, и, соответственно, по-разному организовать свое поведение, системный вызов возвращает в них разные значения. При успешном создании нового процесса в процесс-родитель возвращается положительное значение, равное идентификатору процесса-ребенка. В процесс-ребенок же возвращается значение 0. Если по какой-либо причине создать новый процесс не удалось, то системный вызов вернет в инициировавший его процесс значение -1. Таким образом, общая схема организации различной работы процесса-ребенка и процесса-родителя выглядит так:

```
pid = fork();
if(pid == -1){

...
/* ошибка */
...
} else if (pid == 0){

...
/* ребенок */
...

} else {
...
/* родитель */
...

}
```

Завершение процесса. Функция exit()

Существует два способа корректного завершения процесса в программах, написанных на языке С. Первый способ мы использовали до сих пор: процесс корректно завершался по достижении конца функции main() или при выполнении оператора return в функции main(), второй способ применяется при необходимости завершить процесс в каком-либо другом месте программы. Для этого используется функция exit() из стандартной библиотеки функций для языка С. При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков, после чего инициируется системный вызов прекращения работы процесса и перевода его в состояние закончил исполнение.

Возврата из функции в текущий процесс не происходит и функция ничего не возвращает.

Значение параметра функции exit() – кода завершения процесса – передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс. На самом деле при достижении конца функции main() также неявно вызывается эта функция со значением параметра 0.

Функция для нормального завершения процесса Прототип функции

```
#include <stdlib.h>
void exit(int status);
```

Описание функции

Функция exit служит для нормального завершения процесса. При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков (файлов, pipe, FIFO, сокетов), после чего инициируется системный вызов прекращения работы процесса и перевода его в состояние закончил исполнение

Возврата из функции в текущий процесс не происходит, и функция ничего не возвращает.

Значение параметра status – кода завершения процесса – передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс. При этом используются только младшие 8 бит параметра, так что для кода завершения допустимы значения от 0 до 255. По соглашению, код завершения 0 означает безошибочное завершение процесса.

Если процесс завершает свою работу раньше, чем его родитель, и родитель явно не указал, что он не хочет получать информацию о статусе завершения порожденного процесса, то завершившийся процесс не исчезает из системы окончательно, а остается в состоянии закончил исполнение либо до завершения процесса-родителя, либо до того момента, когда родитель получит эту информацию. Процессы, находящиеся в состоянии закончил исполнение, в операционной системе UNIX принято называть процессами-зомби (zombie, defunct).

Изменение пользовательского контекста процесса. Семейство функций для системного вызова exec()

Для изменения пользовательского контекста процесса применяется системный вызов exec(), который пользователь не может вызвать непосредственно. Вызов exec() заменяет пользовательский контекст текущего процесса на содержимое некоторого исполняемого файла и устанавливает начальные значения регистров процессора (в том числе устанавливает программный счетчик на начало

загружаемой программы). Этот вызов требует для своей работы задания имени исполняемого файла, аргументов командной строки и параметров окружающей среды. Для осуществления вызова программист может воспользоваться одной из шести функций: execlp(), execvp(), execl() и, execv(), execle(), execve(), отличающихся друг от друга представлением параметров, необходимых для работы системного вызова exec().

Функции изменения пользовательского контекста процесса Прототипы функций

```
#include <unistd.h>

int execlp(const char *file,
const char *arg0,
... const char *argN,(char *)NULL)
int execvp(const char *file, char *argv[])

int execl(const char *path,
const char *arg0,
... const char *argN,(char *)NULL)

int execv(const char *path, char *argv[])
int execle(const char *path,
const char *arg0,

... const char *argN,(char *)NULL,

char * envp[])

int execve(const char *path, char *argv[], char *envp[])
```

Описание функций

Для загрузки новой программы в системный контекст текущего процесса используется семейство взаимосвязанных функций, отличающихся друг от друга формой представления параметров.

Аргумент file является указателем на имя файла, который должен быть загружен.

Аргумент path – это указатель на полный путь к файлу, который должен быть загружен.

Аргументы arg0, ..., argN представляют собой указатели на аргументы командной строки. Заметим, что аргумент arg0 должен указывать на имя загружаемого файла. Аргумент argv представляет собой массив из указателей на аргументы командной строки. Начальный элемент массива должен указывать на имя загружаемой программы, а заканчиваться массив должен элементом, содержащим указатель NULL.

Аргумент envp является массивом указателей на параметры окружающей среды, заданные в виде строк "переменная=строка". Последний элемент этого массива должен содержать указатель NULL.

Поскольку вызов функции не изменяет системный контекст текущего процесса, загруженная программа унаследует от загрузившего ее процесса следующие атрибуты:

идентификатор процесса;
идентификатор родительского процесса;
групповой идентификатор процесса;
идентификатор сеанса;
время, оставшееся до возникновения сигнала SIGALRM;
текущую рабочую директорию;
маску создания файлов;
идентификатор пользователя;
групповой идентификатор пользователя;
явное игнорирование сигналов;

таблицу открытых файлов (если для файлового дескриптора не устанавливался признак "закрыть файл при выполнении exec()").

В случае успешного выполнения возврата из функций в программу, осуществившую вызов, не происходит, а управление передается загруженной программе. В случае неудачного выполнения в программу, инициировавшую вызов, возвращается отрицательное значение.

Поскольку системный контекст процесса при вызове exec() остается практическ неизменным, большинство атрибутов процесса, доступных пользователю через системные вызовы (PID, UID, GID, PPID и другие, смысл которых станет понятен по мере углубления наших знаний на дальнейших занятиях), после запуска новой программы также не изменяется.

Важно понимать разницу между системными вызовами fork() и exec(). Системный вызов fork() создает новый процесс, у которого пользовательский контекст совпадает с пользовательским контекстом процесса-родителя. Системный вызов exec() изменяет пользовательский контекст текущего процесса, не создавая новый процесс.

Задание для выполнения

Написать программу, которая будет реализовывать следующие функции:

1. сразу после запуска получает и сообщает свой ID и ID родительского
2. процесса;
3. перед каждым выводом сообщения об ID процесса и родительского процесса эта информация получается заново;
4. порождает процессы, формируя генеалогическое дерево согласно варианту, сообщая, что "процесс с ID таким-то породил процесс с таким-то ID";
5. перед завершением процесса сообщить, что "процесс с таким-то ID и таким-то ID родителя завершает работу";
6. один из процессов должен вместо себя запустить программу, указанную в варианте задания.

На основании выходной информации программы предыдущего пункта изобразить генеалогическое дерево процессов (с указанием идентификаторов процессов). Объяснить каждое выведенное сообщение и их порядок в предыдущем пункте.

Варианты индивидуальных заданий

столбце fork описано генеалогическое древо процессов: каждая цифра указывает на относительный номер (не путать с pid) процесса, являющегося родителем для данного процесса. Например, строка 01113 означает, что первый процесс не имеет родителя среди ваших процессов (порождается и запускается извне), второй, третий и четвертый – порождены первым, пятый – третьим.

В столбце exec указан номер процесса, выполняющего вызов exec, команды для которого указаны в последнем столбце. Запускайте команду обязательно с какими-либо параметрами.

№	fork	exec	
1	0111335	1	ls
2	0122346	2	ps
3	0112256	3	pwd
4	0111253	4	whoami
5	0112233	5	df
6	0112444	6	ls
7	0111332	7	Ps
8	0122345	1	pwd
9	0112256	2	whoami
10	0111255	3	time
11	0112234	4	ls

12	0112445	5	ps
13	0111335	6	pwd
14	0122346	7	whoami
15	0112255	1	date
16	0111254	2	ls
17	0112233	3	ps
18	0112446	4	pwd
19	0112255	5	whoami
20	0111254	6	free