Практическое занятие №5 «Процессы»

Процессы — это фундаментальная абстракция в системе *Unix*. Любой процесс *UNIX* может, в свою очередь, запускать другие процессы. Это придает среде процессов *UNIX* иерархическую структуру, подобную дереву каталогов файловой системы. На вершине дерева процессов находится единственный управляющий процесс, экземпляр очень важной программы init, которая является предком всех системных и пользовательских процессов.

Система *UNIX* предоставляет программисту набор системных вызовов для создания процессов и управления ими. Если исключить различные средства для межпроцессного взаимодействия, то наиболее важными из оставшихся будут:

- fork — используется для создания нового процесса, дублирующего вызывающий. Вызов fork является основным примитивом создания процессов
- exec — семейство библиотечных процедур и один системный вызов, выполняющих одну и ту же функцию - смену задачи процесса за счет перезаписи его пространства памяти новой программой. Различие между вызовами exec в основном лежит в способе задания списка их аргументов
- wait — этот вызов обеспечивает элементарную синхронизацию процессов. Он позволяет процессу ожидать завершения другого процесса, обычно логически связанного с ним
- exit — используется для завершения процесса (мы уже рассмотрели на прошлом занятии).

На данном занятии рассмотрим, что представляют собой процессы *UNIX* в целом и рассмотрим назначение и использование приведенных выше системных вызовов.

Получение информации о процессе

Атрибуты процесса

С каждым процессом *UNIX* связан набор атрибутов, которые помогают операционный системе управлять выполнением и планированием процессов, обеспечивать защиту файловой системы. Одной из наиболее важных характеристик является *идентификатор процесса* (*PID — Process IDentifier*).)

Операционная система присваивает каждому процессу неотрицательное число, которое и является идентификатором процесса. В любой момент времени идентификатор процесса уникален, хотя после завершения процесса он может снова быть использован для другого процесса. Некоторые идентификаторы процесса зарезервированы системой для особых процессов. Так, например, процесс с идентификатором 0, называется *планировщиком* (*scheduler*), процесс с идентификатором 1 — это процесс инициализации, выполняющий программу /etc/init. Этот процесс, явно или неявно, является предком всех других процессов в системе *UNIX*.

Кроме того, с процессом связано еще одно число — важным параметром является *идентификатор родительского процесса* (*PPID — Parent Process Identifier*).

Именно эти атрибуты указаны при выводе информации о процессе командой ps.

Есть и другие характеристики, связанные с процессом, о которых мы подробно поговорим позже.

Рассмотрим различные API, предназначенные для запроса атрибутов процессов. Для

использования данных функций следует включить в программу заголовочный файл unistd.h:

```
pid_t getpid(void);
pid_t getppid(void);
pid_t getpgrp(void);
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
```

Типы данных pid_t, uid_t и gid_t являются целыми числами, размерность которых зависит от реализации. Чтобы использовать эти типы, нужно включить в программу заголовочный файл sys/types.h.

Функции getpid и getppid возвращают идентификаторы текущего и родительского процессов соответственно.

Функция getpgrp возвращает идентификатор группы вызывающего процесса.

Функции getuid и getgid возвращают соответственно реальный идентификатор владельца и реальный идентификатор группы вызывающего процесса. Реальные идентификаторы группы и владельца являются идентификаторами лица, создавшего этот процесс.

Функции geteuid и getegid возвращают значения атрибутов *eUID* и *eGID* вызывающего процесса. Эти идентификаторы используются ядром для определения прав вызывающего процесса на доступ к файлам. Такие атрибуты присваиваются также идентификаторам группы и владельца для файлов, создаваемых процессом. В нормальных условиях эффективный идентификатор пользователя процесса совпадает с его реальным идентификатором. Однако в случае, когда установлен бит смены идентификатора пользователя *set-UID* выполняемого файла, эффективный идентификатор владельца процесса будет равен идентификатору владельца исполняемого файла. Это дает процессу такие же права доступа, какими обладает пользователь, владеющий исполняемым файлом. Аналогичный механизм применим и к эффективному идентификатору группы. Так, эффективный идентификатор группы процесса будет отличаться от реального идентификатора группы, если установлен бит смены идентификатора группы *set-GID* соответствующего исполняемого файла.

Задание №1

Написать функцию, которая выводит в стандартный поток вывода информацию о процессе. *Дополнительное задание*: предусмотреть аргумент, с помощью которого можно управлять той информацией, что выводится в стандартный поток вывода.

Создание процессов

Системный вызов fork()

Основным средством для создания процессов является системный вызов fork. Он является механизмом, который превращает *UNIX* в многозадачную систему.

Описание

```
#include <sys/types.h>
```

```
#include <unistd.h>

pid_t fork(void);
```

К результате успешного вызова fork ядро создаст новый процесс, который является почти точной копией вызывающего процесса. Другими словами, новый процесс выполняет копию той же программы, что и создавший его процесс, при этом все его объекты данных имеют те же самые значения, что и в вызывающем процессе.

Созданный процесс называется дочерним процессом (*child process*), а процесс, осуществивший вызов fork, называется родителем (*parent*).

После вызова родительский процесс и его вновь созданный потомок выполняются одновременно, при этом оба процесса продолжают выполнение с оператора, который следует сразу же за вызовом fork.

Вызов fork не имеет аргументов и возвращает идентификатор процесса pid_t. В файле <sys/types.h> определен специальный тип pid_t, который обычно является целым. Пример вызова:

```
#include <sys/types.h>
#include <unistd.h>

pid = fork();
```

Родитель и потомок отличаются значением переменной pid. В родительском процессе значение переменной pid будет ненулевым положительным числом, для потомка же оно равно нулю. Так как возвращаемые в родительском и дочернем процессе значения различаются, то программист может задавать различные действия для двух процессов.

Значение, возвращаемое родительскому процессу в переменной pid, является идентификатором процесса (*process id*) дочернего процесса. Это число идентифицирует процесс в системе аналогично идентификатору пользователя. Поскольку все процессы порождаются при помощи вызова fork, то каждый процесс *UNIX* имеет уникальный идентификатор процесса. Отрицательное, а на самом деле равное -1, значение переменной pid, возвращается, если вызову fork не удается создать дочерний процесс. В этом случае переменная переменная errno содержит один из двух кодов ошибки:

- EAGAIN — количество текущих процессов в системе превышает установленное системой ограничение, попытайтесь выполнить вызов позже;
- ENOMEM — для создания нового процесса не хватает свободной памяти.

Это может означать, что либо для создания нового процесса не хватает свободной памяти, либо вызывающий процесс попытался нарушить одно из двух ограничений: первое из них - системное ограничение ни число процессов; второе ограничивает число процессов, одновременно выполняющихся и запущенных одним пользователем. Необходимо отметить, что существуют устанавливаемые системой ограничения на максимальное количество процессов, создаваемых одним пользователем (CHILD_MAX), и максимальное количество процессов, одновременно существующих во всей системе (MAXPID). Если любое из этих ограничений при вызове fork нарушается, то функция возвращает код неудачного завершения. Символы MAXPID и CHILD_MAX определяются соответственно в заголовках <sys/param.h> и <limits.h>. Кроме того, процесс может получить значение CHILD_MAX с помощью функции sysconf():

```
int child_max = sysconf(_SC_CHILD_MAX);
```

При успешном вызове fork создается порожденный процесс. Как порожденный процесс, так и родительский планируются ядром *UNIX* для выполнения независимо, а очередность запуска этих процессов зависит от реализации *ОС*. Следует обратить внимание на то, что поскольку оба процесса, созданных программой, будут выполниться одновременно без синхронизации, то нет гарантии, что вывод родительского и дочернего процессов не будет смешиваться. Таким образом, родительский и порожденный процессы могут выполнять различные задачи одновременно.

Пример использования:

```
#include <sys/types.h>
#include <unistd.h>

int main(void) {
	pid_t pid;   /* идентификатор процесса */
	printf("Пока всего один процесс\n");
	printf("Вызов функции fork...\n");
	pid = fork();  /* Создание нового процесса */
	if (pid == 0) {
		printf("Процесс - потомок\n");
	} else if (pid > 0) {
		printf("Родительский процесс, pid потомка %d\n", pid);
	} else {
		printf("Ошибка вызова fork, потомок не создан\n"); }
	}
	return 0;
}
```

Задание №2

Создать один процесс — потомок. Продемонстрировать «непредсказуемость» алгоритма переключения процессов. С помощью функции time() (получение количества секунд, прошедших с начала эры *UNIX*, заголовочный файл time.h, вызов: time_t now = time(NULL);) заставить проработать программу заданное количество секунд (2, 3, 5, … можно указать с помощью макроопределения) и просчитать, сколько раз в каждом процессе выполнится тело цикла — увеличения счетчика.

Задание №3

В программе функция fork может быть вызвана более одного раза. Ее можно вызывать как из родительского потока, так и из потоков — потомков. Родительский процесс содержит локальную переменную. Напишите программу, которая создает два под-процесса, а каждый из них, в свою очередь, свои два под-процесса. После каждого вызова fork в новом процессе увеличить значение локальной переменной, вывести на экран с помощью функции printf значение этой локальной переменной, ее адрес и идентификаторы родительского и дочернего процессов. Кроме того, каждый родительский процесс должен вывести идентификаторы своих дочерних процессов. Обратить внимание на очередность создания процессов при каждом

запуске программы.

Семейство вызовов exec()

В операционный системе *UNIX* программный запуск нового процесса — программы, разделяется на два этапа: создание нового процесса (разветвление, *forking*, вызов fork) и загрузка в память и исполнение нового образа процесса (*process image*). Существует системный вызов (а также, целый набор построенных на нем функций), предназначенный для загрузки двоичной программы в память. Этот новый двоичный образ заменяет предыдущее содержимое адресного пространства, и начинает выполнение новой программы. Такое выполнение (*executing*) новой программы обеспечивается семейством функций exec. Основное отличие между разными функциями из этого семейства состоит в количество и способе передачи параметров новой программе. Следует отметить, что в конечном итоге все эти функции выполняют один системный вызов execve.

Системный вызов execve() загружает в процесс другую программу и передает ей безвозвратное управление. Этот системный вызов объявлен в заголовочном файле unistd.h таким образом:

```
int execve(const char *path, const char **argv, const char **env);
```

Данный системный вызов execve() принимает три аргумента:

- path — это путь к исполняемому файлу программы, которая будет запускаться внутри процесса. Здесь следует учитывать, что переменная окружения PATH в системном вызове execve() не используется. Таким образом, ответственность за поиск программы возложена на автора программы.
- argv — это массив строк — аргументов программы. Следует помнить, что первый аргумент (argv[0]) этого массива может быть либо именем программы, либо чем-то другим (на усмотрение программиста), но не фактическим аргументом. Последним элементом массива argv должен быть NULL.
- env — это массив строк, содержащий окружение запускаемой программы. Этот массив также должен заканчиваться элементом NULL.

Выполняющийся внутри процесса код называется образом процесса (*process image*). Системный вызов execve() заменяет текущий .образ процесса на новый. Таким образом, в случае успешного вызова, возврата в исходную программу не происходит, поскольку работа исходной программы в текущем процессе на этом заканчивается. В случае ошибки execve() возвращает -1.

Таким образом, возврат из системного вызова execve() происходит только в том случае, если произошла ошибка. А так как execve() не может возвращать ничего кроме -1, то можно не выполнять приведенную проверку:

```
if (execve(binary_file, argv, environ) == -1) {
    /* обработка ошибки */
}
```

Можно применить более простую форму обнаружения ошибки:

```
execve(binary_file, argv, environ);
/* обработка ошибки */
```

Рассмотрим пример применения данной функции:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (void) {
        extern char **environ;
        char * uname_args[] = { "uname", "-a", NULL };

        execve("/bin/uname", uname_args, environ);
        fprintf (stderr, "Execve error\n");

        return EXIT_SUCCESS;
}
```

В данной программе образ текущего процесса был заменен программой /bin/uname с опцией -a. Если программа была успешно вызвана (независимо от того, как она завершилась), то сообщение Execve error не выводится. Следует отметить, что с помощью функции execve программе можно создать любое окружение. В частности, в элементе массива argv[0] может находиться все что угодно. Запускаемой программе можно передавать пустое окружение. Для этого следует указать NULL в третьем аргументе системного вызова execve.

```
execve("/bin/uname", uname_args, NULL);
```

В стандартной библиотеке языка *Cи* есть пять дополнительных функций, которые реализованы с использованием системного вызова execve() и часто все вместе называются семейством exec(). Все функции семейства exec() объявлены в заголовочном файле unistd.h следующим образом:

```
int execl(const char    *path, const char *arg0, ...);
int execle(const char  *path, const char *arg0, ..., const char **env);
int execlp(const char  *file,   const char *arg0, ...);
int execv(const char   *path, const char **argv);
int execvp(const char *file,   const char  **argv);
```

Названия функций соответствуют простому мнемоническому правилу: буквы l и v в названии указывают, что передаются аргументы в виде списка (*list*) или массива (*vector*), соответственно; буква p обозначает, что поиск указанного файла осуществляется по полному пользовательскому пути (path). В командах, которые указываются при вызове функций с буквой p в имени, можно указывать только имя файла, если этот файл находится в каталогах, пользовательской переменной PATH. Наконец, буква e в названии указывает на то, что для нового процесса также передается новые переменные окружения. Если же буквы e в названии нет, то начальным окружением запускаемой программы будет окружение текущего процесса.

Следует еще раз подчеркнуть, что вызов любой функции из семейства exec не создает новый под-процесс, который выполняется одновременно с вызывающим, а вместо этого новая программа загружается на место старой. Поэтому, в отличие от вызова fork, успешный вызов exec не возвращает значения. В случае ошибки вызов любой функции из семейства exec возвращают значение -1 и присваивают переменной errno одно из следующих значений:

E2BIG

Общее число байтов в предоставленном списке аргументов (arg) или среде (env) слишком велико.

EACCESS

У процесса отсутствует разрешение на поиск для компонента пути, указанного при помощи аргумента path; path не является обычным файлом; целевой файл не помечен как исполняемый; или файловая система, где находится path или file, подмонтирована с флагом noexec.

EFAULT

Указатель недопустим.

EIO

Произошла низкоуровневая ошибка ввода-вывода.

EISDIR

Последний компонент в пути path, или интерпретатор, является каталогом.

ELOOP

При разрешении пути path встретилось слишком много символических ссылок.

EMFILE

Для вызывающего процесса был достигнут предел числа открытых файлов.

ENFILE

Был достигнут системный предел числа открытых файлов.

ENOENT

Цель path или file не существует, либо не существует необходимая общая библиотека.

ENOEXEC

Цель path или file является недопустимым двоичным файлом, или же она предназначена для другой машинной архитектуры.

ENOMEM

Недостаточно памяти ядра для выполнения новой программы.

ENOTDIR

Один из компонентов в пути path не является каталогом.

EPERM

Файловая система, которой принадлежит path или file, подмонтирована с флагом nosuid, пользователь не является пользователем root, либо для path или file установлен бит suid или sgid.

ETXTBSY

Цель path или file открыта для записи другим процессом.

Чаще всего системные вызовы fork и execve используются совместно, что позволяет запускать программы в отдельных процессах.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main (void) {
	pid_t result;
	extern char **environ;
	char *sleep_args[] = { "sleep", "5", NULL };

	result = fork ();
	if (result == -1) {
		fprintf (stderr, "Fork Error\n");
		exit(EXIT_FAILURE);
	}

	if (result ==0) {
		execve("/bin/sleep", sleep_args, environ);
		fprintf (stderr, "Execve Error\n");
		exit(EXIT_FAILURE);
	} else {
		fprintf(stderr, "Parent process. PID=%d\n", getpid());
	}

	return EXIT_SUCCESS;
}
```

Задание №4

Предположим, что вызовы execvp и execlp не существуют. Напишите эквиваленты этих процедур, используя вызовы execl и execv. Параметры этих процедур должны состоять из списка каталогов и набора аргументов командной строки.

Синхронизация процессов

Процессы в *UNIX* работают независимо друг от друга. Но иногда необходимо организовать последовательное выполнения процессов, причем часто нужно сделать так, чтобы процесс — потомок после своего завершения должен передать некоторую информацию о

результате своего выполнения в родительский процесс. Разработчиками *UNIX* было решено, что, когда процесс умирает раньше своего предка, ядро должно переводить потомок в особое состояние. Процесс в таком состоянии называется зомби (*zombie*). В этом состоянии существует только минимальный скелет того, что раньше было процессом, — некоторые базовые структуры данных ядра, содержащие потенциально полезные данные. Процесс в этом состоянии ожидает того, что родитель запросит его статус (это называется обслуживанием (*waiting on*) процесса-зомби). Только после того, как предок получает сохраненную информацию о завершенном потомке, процесс формально удаляется и прекращает существовать даже в виде зомби.

Функция wait()

Ядро *UNIX* систем предоставляет несколько функций для получения информации о завершенных дочерних процессах. Одна из этих функций — это wait():

```
#include <sys/types.h>
#include <sys/wait.h> /* можно #include <wait.h> */

pid_t wait(int *status);
```

Этот системный вызов блокирует родительский процесс до тех пор, пока не завершится один из его потомков. Таким образом, родительский процесс временно приостанавливает свое выполнение, в то время как дочерний процесс продолжает выполняться. После завершения дочернего процесса выполнение родительского процесса продолжится. Если запущено более одного дочернего процесса, то возврат из вызова wait произойдет после выхода из любого из потомков.

Вызов wait() возвращает идентификатор pid завершенного дочернего процесса или значение -1, если произошла ошибка. Если ни один из дочерних процессов пока что не завершился, вызов блокируется до тех пор, пока это событие не произойдет. Если дочерний процесс уже завершился, то вызов возвращает результат немедленно, без блокировки.

В случае ошибки переменной errno присваивается одно из двух значений:

ECHILD

У вызывающего процесса нет дочерних процессов.

EINTR

Во время ожидания был получен сигнал, и вызов вернул результат раньше времени.

Вызов wait принимает один аргумент, status — указатель на целое число. Если указатель равен NULL, то аргумент просто игнорируется. Если же вызову wait передается допустимый указатель, то после возврата из вызова wait переменная status будет содержать полезную информацию о статусе завершения процесса. Обычно эта информация будет представлять собой код завершения процесса — потомка, переданный при помощи вызова exit.

Для извлечения этой информации существуют специальные макросы:

```
#include <sys/wait.h>  /* или <wait.h> */

int WIFEXITED(status);
int WIFSIGNALED(status);
int WIFSTOPPED(status);
int WIFCONTINUED(status);
```

```
int WEXITSTATUS(status);
int WTERMSIG(status);
int WSTOPSIG(status);
int WCOREDUMP(status);
```

Пока мы рассмотрим только некоторые из этих макросов: WIFEXITED — возвращает ненулевое значение, если процесс — потомок завершится обычным образом: в результате возврата из функции main() или в результате вызова exit(). В этом случае макрос WEXITSTATUS предоставляет восемь бит младших разрядов, которые были переданы exit(). Макрос WIFSIGNALED возвращает ненулевое значение, если процесс был завершен в результате получения сигнала. Об этом и об остальных макросах поговорим на следующих занятиях.

Рассмотрим пример применения вызова wait.

```
#include <stdio.h>
#include <stdlib.h>
#include <wait.h>
#include <unistd.h>
#include <sys/types.h>

int main(void) {
	pid_t status, childpid;
	int exit_status;

	status = fork();
	if {status == -1) {
		fprintf (stderr, "Fork Error\n");
		exit(EXIT_FAILURE);
	}

	/* Потомок */
	if (status == 0) {
		execlp("sleep", "sleep", "30", NULL);
		fprintf(stderr, "Exec Error\n");
		exit(EXIT_FAILURE);
	}

	/* Родитель */
	childpid = wait(&exit_status);
	if( childpid == -1) {
		fprintf(stderr, "Wait Error\n");
		exit(EXIT_FAILURE);
	}

	if (WIFEXITED(exit_status)) {
		printf ("Process with PID=%d "
			"has exited with code=%d\n", childpid,
```

```
                WEXITSTATUS(exit_status));
        }

        if (WIFSIGNALED(exit_status)) {
                printf ("Process with PID=%d "
                        "has exited with signal \n", childpid);
        }

        return EXIT_SUCCESS;
}
```

Ожидание завершения определенного потомка

Системный вызов wait позволяет родительскому процессу ожидать завершения любого дочернего процесса. Достаточно часто у процессов бывает несколько потомков и процессу нужно дождаться завершения не всех их, а какого-то конкретного процесса — потомка. В принципе, можно несколько раз сделать системный вызов wait, каждый раз проверяя возвращаемое значение. Однако это очень неудобно. Если известно значение идентификатора pid интересующего вас процесса, то можно воспользоваться системным вызовом waitpid():

```
#include <sys/types.h>
#include <sys/wait.h>   /* или <wait.h> */

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

При помощи параметра pid можно указать, какой процесс или какие процессы нужно ожидать. Он может принимать значения из четырех категорий:

< -1

Предписывает ожидать любые дочерние процессы, значение идентификатора группы процессов для которых равно абсолютному значению данного параметра.

-1

Предписывает ожидать любые дочерние процессы. Поведение, аналогичное поведению системного вызова wait.

0

Предписывает ожидать любые дочерние процессы, принадлежащие той же группе процессов, что и вызывающий процесс.

> 0

Предписывает ожидать дочерний процесс, значение идентификатора pid которого в точности равно значению данного параметра.

Параметр status аналогичен единственному параметру системного вызова wait(), и его можно использовать с описанными выше макросами.

В качестве значения параметра options можно передавать опции: определенные значения, при необходимости объединенные операцией двоичного ИЛИ, или же пустое значение:

WNOHANG

Вызов не должен блокироваться. Если уже завершенного (или остановленного, или

продолженного) подходящего дочернего процесса нет, вызов должен возвращать результат немедленно.

WUNTRACED

Если это значение передается системному вызову, то устанавливается флаг WIFSTOPPED, даже если вызывающий процесс не отслеживает дочерний процесс. Этот флаг помогает реализовывать более общее управление заданиями, как, например, в оболочке.

WCONTINUED

Если системному вызову передается это значение, то устанавливается флаг WIFCONTINUED, даже если вызывающий процесс не отслеживает дочерний процесс. Как и WUNTRACED, этот флаг полезен для реализации оболочки.

В случае успешного завершения системный вызов waitpid() возвращает идентификатор pid процесса, состояние которого изменилось. Если был передан флаг WNOHANG, а указанный потомок или потомки еще не изменили состояние, то вызов waitpid() возвращает значение 0. В случае ошибки вызов возвращает -1 и присваивает переменной errno одно из трех значений:

ECHILD

Процесс или процессы, указанные при помощи аргумента pid, не существуют либо они не являются потомками вызывающего процесса.

EINTR

Параметр WNOHANG не был указан, и во время ожидания был получен сигнал.

EINVAL

Аргумент options имеет недопустимое значение.

Следует обратить внимание на то, что такая форма системного вызова wait():

```
wait(&status);
```

идентична такому варианту системного вызова waitpid():

```
waitpid(-1, &status, 0);
```

Рассмотрим пример применение waitpid.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <wait.h>

#define DOT_COUNT     15

int main (void) {
        int i, status;
```

```
        /* Child */
        if (!fork()) {
                for (i = 0; i < DOT_COUNT; i++) {
                        fprintf(stderr, ".");
                        sleep(1);
                }
                exit(5);
        }

        /* Parent */
        while (1) {
                if (!waitpid(-1, &status, WNOHANG)) {
                        fprintf(stderr, "*");
                else {
                        fprintf(stderr, "(exit)\n");
                        break;
                }
                sleep(2);
        }

        if (WIFEXITED (status)) {
                fprintf(stderr, "Exited with code="
                        "%d\n", WEXITSTATUS(status));
        } else if (WIFSIGNALED (status)) {
                fprintf(stderr, "Exited by signal\n");
        }

        return EXIT_SUCCESS;
}
```

Здесь системный вызов fork() создает новый процесс, в котором выполняется цикл ежесекундного вывода точек на экран. После завершения этого цикла вызывается функция exit(), которая завершает не программу, а текущий процесс. Аргумент 5 здесь выбран произвольно.

В это время процесс — родитель каждые 2 секунды вызывает waitpid(). Первый аргумент этого системного вызова равен -1, что означает ожидание завершения любого дочернего процесса. Второй аргумент — это статус завершившегося потомка. Третий аргумент содержит флаг WNOHANG, благодаря чему родительский процесс не блокируется в ожидании завершения дочернего процесса. Если потомок еще не завершился, то waitpid() просто возвращает 0. И так каждые 2 секунды до тех пор, пока дочерний процесс не завершится. После этого происходит исследование переменной status.

Запустить программу. Рассмотреть ситуации, когда программа нормально завершает свою работу, и когда она прерывается командой KILL.

Задание № 5

С помощью функций fork, exec, wait создайте упрощенный аналог функции system() из

прошлого занятия.

Процессы зомби

Когда процесс завершается, он остается в состоянии «зомби» (*zombie*) до тех пор, пока окончательно не будет «вытерт» родительским процессом. В этом режиме единственным занимаемым ресурсом остается специальная структура в памяти ядра, в которой хранится статус выхода, а также информация об использовании ресурсов системы. Эта информация может быть важна для родительского процесса, который получает ее посредством вызова wait, который также освобождает данную структуру в системной памяти процесса — потомка. Если родительский процесс завершается раньше, чем его потомок, то тот усыновляется процессом init. После завершения работы потомка процесс init вызовет wait для освобождения его структуры данных.

Определенная проблема возникает в том случае, если процесс завершится раньше своего родителя и последний не вызовет wait. Тогда структура процесса-потомка не будет освобождена, а потомок останется в состоянии зомби до тех пор, пока система не будет перезагружена. Такая ситуация возникает очень редко, так как разработчики стараются не допустить ее возникновения в своих программах. Однако потенциальная возможность такой ситуации остается, когда недостаточно внимательно написанные программы не следят за всеми своими процессами-потомками. Это может быть достаточно неприятной ситуацией, так как процессы-зомби видимы при помощи ps, а пользователи никак не могут их завершить, так как они уже завершены. Более того, они продолжают занимать структуру в памяти процесса, уменьшая тем самым максимальное количество активных процессов.

Рассмотрим пример, временно создающий процесс — зомби.

```
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int main (void) {
	int status;

	/* Child */
	if («fork ()) {
		execlp("ls", "ls", NULL);
		fprintf (stderr, "Exec error\n");
	}

	/* Parent */
	sleep(40);
	wait(&status);

	if (WIFEXITED (status)) {
		printf ("Code=%d\n", WEXITSTATUS(status));
	}

	return 0;
}
```

В промежуток времени между завершением дочернего процесса и вызовом функции wait() (40 секундный «сон» родителя) в системе будет существовать зомби. С помощью команды ps -l это можно увидеть.

Задание №6

Напишите аналогичную программу и посмотрите на созданый процесс — зомби.

Задание №7

Рассмотрим простейший способ организовать взаимодействие процессов: родителя и потомка. Родительский процесс может передавать своему потомку некоторые данные с помощью аргументов одной из функций семейства exec(). В свою очередь дочерний процесс при нормальном завершении сообщает родителю свой код возврата. Рассмотрим задачу, в которой попробуем осуществить такое взаимодействие.

С помощью методы Монте – Карло вычислить площадь круга заданного радиуса и оценить значение числа π.

Суть метода Монте – Карло: пусть в прямоугольнике на плоскости $a \le x \le b$, $c \le y \le d$ задана некоторая фигура M и требуется найти ее площадь. В прямоугольнике выбирается n случайных точек («бросков»); пусть m из них попало в область M. Площадь фигуры оценивается как:

$$S_M \approx \frac{m}{n} \cdot S_p$$, где $S_p = (b-a)\cdot(c-d)$ — площадь прямоугольника.

Точность оценки будет тем выше, чем больше число бросаний n. Погрешность уменьшается квадратично с ростом числа испытаний.

Написать программу, которая через аргументы командной строки получает натуральное число — количество «бросков», а через код возврата возвращает родителю количество точек, попавших в заданную область. Вторая программа запускает процесс — потомок, в него загружает эту программу, получает результат и с его помощью вычисляет требуемые характеристики.

Написать программу с одним родительским процессом и одним процессам — потомком.

Дополнительное задание — обобщить задачу на n потоков и усреднить результаты отдельных n «экспериментов». Параметры (количество бросков и количество потоков) передаются с помощью опций.