1. Каким образом может быть порожден новый процесс? Какова структура нового процесса?

Для порождения нового процесса (процесс-потомок) используется системный вызов fork(). Формат вызова:

int fork();

Порожденный таким образом процесс представляет собой точную копию своего процесса-предка. Единственное различие между ними заключается в том, что процесс-потомок в качестве возвращаемого значения системного вызова fork() получает 0, а процесс-предок – идентификатор процесса-потомка. Кроме того, процесс-потомок наследует и весь контекст программной среды, включая дескрипторы файлов, каналы и т.д. Наличие у процесса идентификатора дает возможность и ОС UNIX, и любому другому пользовательскому процессу получить информацию о функционирующих в данный момент процессах.

2. Если процесс-предок открывает файл, а затем порождает процесс-потомок, а тот, в свою очередь, изменяет положение указателя чтения-записи файла, то изменится ли положение указателя чтения-записи файла процесса-отца?

Да, изменится.

С таблицей описателей файлов тесно связана таблица файлов. Каждый элемент таблицы файлов содержит информацию о режиме открытия файла, специфицированным при открытии файла, а также информацию о положении указателя чтения-записи. При каждом открытии файла в таблице файлов появляется новый элемент.

Один и тот же файл ОС UNIX может быть открыт несколькими не связанными друг с другом процессами, при этом ему будет соответствовать один элемент таблицы описателей файлов и столько элементов таблицы файлов, сколько раз этот файл был открыт.

Однако из этого правила есть одно исключение: оно касается случая, когда файл, открытый процессом, потом открывается процессом-потомком, порожденным с помощью системного вызова fork(). При возникновении такой ситуации операции открытия файла, осуществленной процессом-потомком, будет поставлен в соответствие тот из существующих элементов таблицы файлов (в том числе положение указателя чтения-записи), который в свое время был поставлен в соответствие операции открытия этого файла, осуществленной процессом-предком.

3. Что произойдет, если процесс-потомок завершится раньше, чем процесспредок осуществит системный вызов wait()?

До момента вызова wait() ничего не произойдет.

В тот момент, когда процесс-отец получает информацию о причине смерти потомка, паспорт умершего процесса наконец вычеркивается из таблицы процессов и может быть переиспользован новым процессом. До того он хранится в таблице процессов в состоянии "zombie" - "живой мертвец". Только для того, чтобы кто-нибудь мог узнать статус его завершения.

4. Могут ли родственные процессы разделять общую память?

Да.

Процессы могут разделять один и тот же открытый файл. Для исключения конфликтов, возникающих при попытке одновременной записи в одно и то же место в нем, применяется механизм блокировки. Процесс может блокировать файл целиком или отдельную его часть (запись). Под записью здесь подразумевается непрерывная область набора данных с указанными началом и длиной.

Перед выполнением операции записи в файл, процесс должен проверить, не блокирована ли запись, в которую предполагается поместить новые данные. Если область блокирована другим процессом, первый должен ждать ее освобождения. Затем надо блокировать запись и произвести операцию вывода. По завершении действий блокировка отменяется, давая другим процессам возможность записывать свои данные в этот файл. Для выполнения операции чтения блокировку можно не делать.

Для блокирования записи используется системный вызов lockf( ):

```
#include <unistd.h>

int lockf (int fd, int function, long size);
```

Здесь *fd* - дескриптор файла. Аргумент function задает выполняемую операцию и может принимать следующие значения:

F_ULOCK	- отменить предыдущую блокировку;
F_LOCK	- блокировать запись;
F_TLOCK	- блокировать запись с проверкой, не блокирована ли она другим процессом;
F_TEST	- проверить, не блокирована ли запись другим процессом.

Начало записи определяется текущим положением указателя в файле. Длина записи задается аргументом size. При неудачной попытке блокирования записи функция возвращает в качестве своего значения (-1).

Обмен данными между процессами через разделяемые файлы имеет ряд недостатков. Основным из них является то, что операции записи и чтения в файлы на диске требуют для своего выполнения гораздо больше времени, чем перемещение данных в оперативной памяти.

5. Каков алгоритм системного вызова fork()?

1. Проверить доступность ресурсов ядра.

2. Получить свободное место в таблице процессов и уникальный идентификатор процесса.

3. Проверить, не запустил ли пользователь слишком много процессов (не превышено ли ограничение).

4. Сделать пометку, что порождённый процесс находится в состоянии создания.

5. Скопировать информацию в таблицу процессов из записи, соответствующей родительскому процессу, в запись, соответствующую порождаемому процессу.

6. Увеличить значение счётчика открытых файлов в таблице файлов.

7. Сделать копию контекста родительского процесса

8. Если в данный момент выполняется родительский процесс, то:

- перевести порождаемый процесс в состояние готовности;

- возвратить идентификатор процесса.

9. Если выполняется порождённый процесс, то:

- записать начальные значения в поля синхронизации адресного пространства

6. Какова структура таблиц открытых файлов, файлов и описателей файлов после создания процесса?

таблица описателей файлов = 0

таблица файлов = 0

таблица открытых файлов процесса = 0

7. Каков алгоритм системного вызова exit()?

1. Игнорировать все сигналы.

2. Закрыть все открытые файлы.

3. Освободить области и память, ассоциированные с процессом.

4. Создать запись учётной информации.

5. Прекратить существование процесса.

6. Назначить всем процессам-потомкам в качестве родителя процесс init().

7. Если какой-либо из потомков прекратил существование, то послать процессу init сигнал

гибели потомка.

8. Переключить контекст.

8. Каков алгоритм системного вызова wait()?

Ожидание завершения процесса-потомка родительским процессом выполняется с помощью системного вызова wait()

int wait(int *status);

1. Если процесс, который вызвал wait, не имеет потомков, то возвратить код ошибки.

2. В бесконечном цикле:

* Если процесс, вызвавший wait, имеет потомков, прекративших существование:

- выбрать произвольного потомка;

- передать его родителю информацию об использовании потомком ресурсов;

- освободить в таблице процессов место, занятое процессом;

- выдать идентификатор процесса, код возврата status из системного вызова exit,

вызванного потомком.

* Приостановиться с приоритетом, допускающим прерывание, до завершения потомка.

9. В чем разница между различными формами системных вызовов типа exec()?

Запуск программ, находящихся в отдельных файлах, в рамках текущего процесса возможен при использовании следующих системных вызовов:

int execl (char *path, char* arg1, ...)

Запуск программы path с параметрами arg1, arg2… Последний параметр должен иметь значение NULL.

int execv (char *path, char* argv[])

Запуск программы path с параметрами argv[i]. Последний параметр должен иметь значение NULL.

int execle(path, arg0, arg1, ..., argn, 0, envp)

Запуск программы path с параметрами arg1, arg2…

позволяют вызывающему назначить окружение исполняемой программе через параметр envp. Аргумент envp является массивом указателей на строки (завершающиеся null), он должен заканчиваться указателем null.

int execve(path, argv, envp);

Запуск программы path с параметрами argv[i]. Последний параметр должен иметь значение NULL.

envp — это массив строк в формате ключ=значение, которые передаются новой программе в качестве окружения (environment).

int execlp (char *file, char *arg1, ...);

Запуск программы file с параметрами arg1, arg2… Последний параметр должен иметь значение NULL. Если file не содержит символов /, поиск программы ведётся, используя переменную окружения PATH.

int execvp (char *file, char* argv[]);

Запуск программы file с параметрами argv[i]. Последний параметр должен иметь значение NULL. Если file не содержит символов /, поиск программы ведётся, используя переменную окружения PATH.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

Функции exec() возвращают значение только при возникновении ошибки. При этом возвращается -1, а errno присваивается код ошибки.