Процессы UNIX.

Процесс в UNIX - основная абстракция системы;

Так во всех ОС, но UNIX определяет ОС очень конкретно;

Основная единица декомпозиции ОС, т.к. именно процессу выделяются ресурсы;

Поток - часть кода программы, которая может выполняться параллельно с другими частями кода программы;

Владелец ресурса - процесс!

Поток - единица диспетчеризации;

(Диспетчер (google) - тот, кто координирует действия)

Диспетчеризация - выделение потоку процессорного времени;

Но выделенные ресурсы принадлежат процессу;

С т.з. UNIX процесс часть время выполняется в режиме пользователя (в режиме задачи), тогда он выполняет собственный код, и часть в режиме ядра, тогда он выполняет реентерабельный код ОС;

Реентерабельный код - код повторной входимости (код чистых процедур);

Повторная входимость - сразу несколько процессов

могут вызвать одну и ту же ф-цию ОС, находясь при этом в разных её точках;

Любая ОС должна иметь реентерабельные коды;

Исключение - MS-DOS (создавалась как однозадачная ОС и была плохо спроектирована);

Реентерабельность кодов ОС достигается за счёт того, что в кодах функции нет данных (соответственно при их выполнении ничего не меняется), их выносят из чистых процедур;

В UNIX любой процесс создаётся системным вызовом fork(); (не вилка, а развилка)

Да, есть clone(), system(), но процесс создаётся с помощью fork();

Любой процесс может создать любое кол-во потомков;

Форк-бомба

.....

Вспомним, что это уже было, продолжаем семинар 1

В UNIX процессы объединяются в группы, т.к. процессы одной группы могут получать одни и те же сигналы (но могут и не получать, это устанавливается соотв. флагами);

Вапснейшая группа - терминальная;

Фактически любой запущенный терминал - - процесс;
Все процессы, запущенные на данном терминале, явл. его потомками

Процесс-родитель, вызвавший fork(), становится создателем (лидером) группы;

Вапснейшее событие в системе - завершение процесса:
родитель проанализировать статус завершения потомков;

Флаг copy on write работает в паре с изменёнными правами доступа only read (обычные - read/write - к данным и стеку);

Флаг copy on write - огромное достижение проблемы разделения данных (коллективного использования страниц);

Результат появления флага copy on write:
во всех совр. ОС исп. упр-е памятью страницами по запросу;
Виртуальная память - страничная;

Чтобы не было неконтролируемого разделения одного и того же адресного пр-ва, код прежде копировался в адр. пр-во потомка. В рез-те в системе могло существовать много копий одной и той же программы, что крайне неэффективно;

Появление copy on write позволило предку и потомкам разделять одно адр. пр-во (предка);

Это можно назвать опр. способом взаимо-исключения;

ОС не контролирует, в какой ветке (предок или потомок) вызываются exit() и exec(), т.к. это сильно усложнит код (очень мощные проверки были бы), т.к. потомок - потенциальный родитель;

wait() в child и exec() в предке не синхронизируются (где они вызваны);

ОС не может усложняться искусственно;

exec()

2 лаба - оптимизация fork;

Выучить наизусть часть методички про exec();

Пояснения „своими словами":

exec() переводит процесс (не потомка) на новое адресное пространство программы, которая передана exec() в качестве параметра;

fork() уже создал потомку адр. пр-во, но оно ссылается на адр. пр-во предка;

(дескрипторы страниц, которые его описывают)

Алгоритм работы exec():

1) exec проверяет, существует ли данный файл;

{ имя файла - строка символов;
полное имя файла = путь + имя;

{ В UNIX имя файла начинается с „/" (корневой каталог);
В Windows - с имени логического диска;

Строка проверяется до очередного „/", так ОС проходит каталоги и подкаталоги в поисках файла по указанному пути;

{ 3 типа файлов:
1) исходники;
2) объектники;
3) исполняемые;

2) Исполняемый ли файл?

3) Есть ли соотв. права для запуска?

4) Если всё благополучно, то для программы, которая передаётся exec() в качестве параметра, создаётся адресное пр-во, т.е. создаются карты трансляции адресов (в совр. ОС это таблицы страниц);

Для одного адр. пр-ва создаётся большое кол-во таблиц страниц (а не одна), т.к. в совр. ОС

адр. пр-ва очень большие и для их описания требуется много таблиц;

В совр. компьютерах на базе процессоров Intel в 64-разр. ОС схема преобразования называется PAE (Physical Address Extension) – 4 уровня таблиц страниц;

Для программы создаётся адр. пр-во, но у потомка уже есть адр. пр-во;

struct proc:
- система оперирует связными списками
- дескриптор процесса имеет указатель на таблицы страниц (struct vmspace *)

5) Удалить старые таблицы страниц, указатель на которые содержится в дескрипторе процесса;
6) Поменять указатель на новые таблицы страниц;
7) Загрузить в регистр IP нач. адрес программы (или адрес точки входа программы, или просто точку входа программы);
+ (в процессорах Intel) в регистр CR3 загрузить адрес таблиц страниц из дескриптора процесса (в дескрипторе мы поменяли адрес таблиц страниц, но его ещё надо загрузить в CR3), чтобы можно было выполнять страничное преобразование, которое нужно для адресации кода;

Результат: будет выполняться новая программа;

Если exec был вызван в потомке, то у предка флаг copy on write будет сброшен и права доступа вернутся к read-write;

Зомби

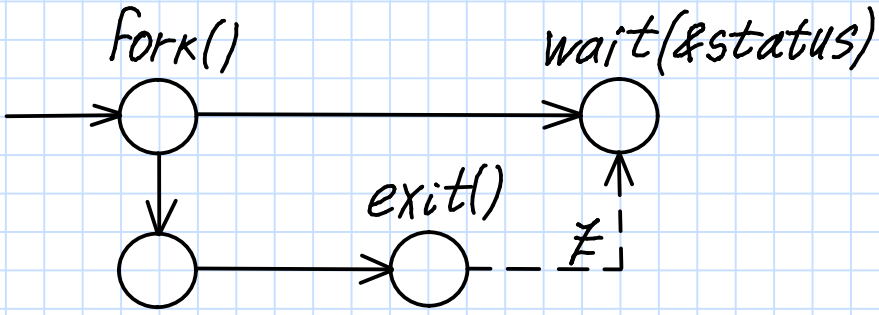
Васильев - „UNIX изнутри"
Есть ещё Роберт Лав, но там больше про практику;

Таблица по wait():

Макрос	Описание
WIFEXITED(stat_val)	Ненулевой, если дочерний процесс завершен нормально
WEXITSTATUS(stat_val)	Если WIFEXITED ненулевой, возвращает код завершения дочернего процесса
WIFSIGNALED(stat_val)	Ненулевой, если дочерний процесс завершается не перехватываемым сигналом
WTERMSIG(stat_val)	Если WIFSIGNALED ненулевой, возвращает номер сигнала
WIFSTOPPED(stat_val)	Ненулевой, если дочерний процесс остановился
WSTOPSIG(stat_val)	Если WIFSTOPPED ненулевой, возвращает номер сигнала

wait() - ожидание завершения потомков, чтобы не возникали процессы-сироты и сохранялась иерархия

Если предок вызвал wait():



Если бы не было Z, то предок будет блокирован на wait() навсегда;

Процесс блокирован ⟹ ему не выделяется процессорное время, но ему всё равно выделены ресурсы: выделены таблицы страниц,

создано адр. пр-во, некоторые страницы могут находиться в памяти (пока ещё не вытеснены), (+таблицы страниц должны частично находиться в обл-ти данных ядра ОС)

а также ему выделен дескриптор;

⟹ блокированные процессы – плохо для ОС;

Поэтому вводится состояние state зомби, чтобы предок, вызвав wait(), смог получить статус завершения своего потомка;

Зомби – процесс, у которого отобраны все ресурсы, кроме последнего – строки в таблице процессов (на самом деле её нет и не может быть);
строка – дескриптор (структура);

↳ Первое, что ОС делает после создания процесса – идентифицирует его (присваивает pid) и выделяет структуру, инициализируя часть полей

В struct proc есть указатель на parent, родственников (siblings) и child;
Иерархия процессов прописана;

Предок вызовет wait(), потомок завершится и предок получит статус его завершения;

---

pipe() („труба")

Основная особенность трубы – 2 конца:
в один конец данные входят, из другого выходят;

Потоковая модель передачи данных, симплексная связь (односторонняя)

2-сторонняя – дуплексная,
для неё нужно 2 трубы

pipe – базовое средство взаимодействия параллельных процессов в UNIX (изначально было включено в сист. вызовы UNIX);

Сист. вызов pipe() создаёт неименованный программный канал (не имеет имени, но имеет дескриптор);

Основная парадигма UNIX: „В UNIX всё файл" (кроме процессов): устройства и прогр. каналы;

Прогр. канал – спец. файл;

В UNIX имя файла не явл. его идентификатором, файл может иметь много разных имён;
Идентификатор файла – номер inode
(inode – дескриптор файл);

Процессы-потомки наследуют дескрипторы открытых файлов
⟹ наследуют и pipe;

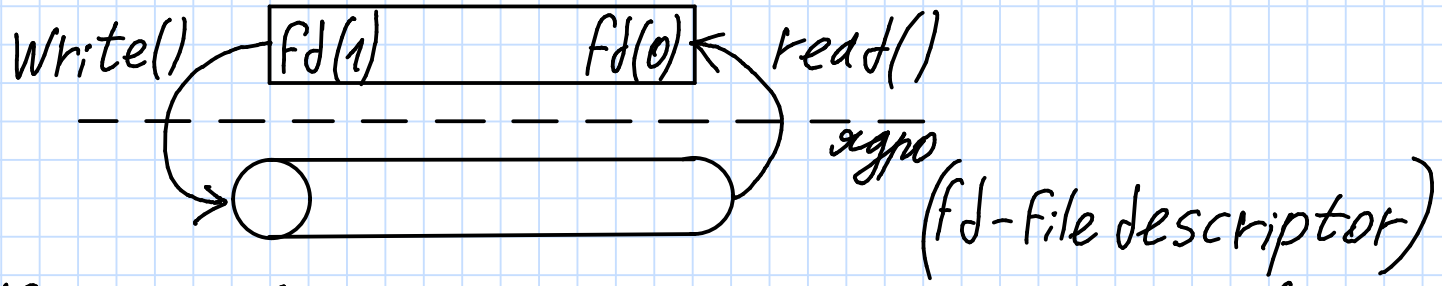
⟹ через pipe могут взаимодействовать только процессы-родственники;

Прогр. канал - буфер в обл-ти данных ядра ОС, т.к. адр. пр-ва процессов явл. защищёнными ⟹ ни один процесс не может обратиться в адр. пр-во др. процесса;

Защита адр. пр-в процессов - основа много-процессности (многозадачности);

Многозадачность - в памяти одновременно находится большое кол-во программ;
Адр. пр-ва этих программ должны быть защищены, иначе ничего работать не будет;

Процесс не может обратиться в адр. пр-во другого процесса, поэтому параллельные процессы могут взаимодействовать только через третье адр. пр-во - адр. пр-во ядра ОС;
Это касается любого взаимодействия процессов;



Именованные прогр. каналы реализованы со встроенным механизмом взаимоисключения: в канал нельзя писать, если из него читают, и из канала нельзя читать, если в него пишут;

```
pid_t childpid;
int fd[2];
if ((childpid = fork()) == -1)
{
    /* errors */
}
else if (childpid == 0)
{
    /* ... */
    close(fd[0]);
    write(fd[1], ...);
}
else
{
    /* ... */
    close(fd[1]);
    read(fd[0], ...);
}
```

канал закрывается на чтение, если в него пишут;
канал закрывается для записи, если из него читают;

ткпоф создаёт именованный программный канал;

В UNIX/Linux поддерживают 2 типа каналов: неименованный и именованный;

Именованный прогр. канал имеет имя, поэтому любой процесс, который знает имя

прогр. канала, можем с ними работать;

---

exec – группа сист. вызовов;
(отличаются набором формальных параметров)

Формальные и фактические параметры, а не аргументы

```
int execl(char const *name, char const *arg0, ..., char const *argn, 0);
int execv(char const *name, char const * const *argv);
int execle(...);
int execve(...);
```

↳ полное имя

признак конца строки параметров

```
// Пример:
int execl("/bin/ls", "ls", "-l", 0);
```

– процесс перейдёт на выполнение команды shell – ls -l;