# Семинар 4 (25.10.2021)

## Unix

Лабораторные мы выполняем на Linux, Linux это UNIX-подобная ОС, поэтому она исполняет все

В UNIX основной абстракцией является процесс. С точки зрения UNIX – процесс часть времени выполняется в режиме задача (или в режиме пользователя) и тогда он выполняет собственный код, а часть времени он выполняется в режиме ядра и тогда он выполняет реентерабельный код операционной системы. В режиме ядра, процесс выполняет повторно входимые процедуры (или процедуры чистого кода). Повторная входимость означает что одну и ту же процедуру могут использовать несколько процессов, при этом они могут находиться … . Такое возможно только с процедурами чистого кода, то есть которые не модифицирует сами себя. Что можно изменить в коде? (Переменную), поэтому вынесены все данные. В ОС-ах существуют специальные системные таблицы (те самые таблицы к которым можно обращаться, какие-то функции ядра могут редактировать, но они не находятся в коде) – это структуры … ядра. То есть в самих чистых процедурах никаких переменных быть не должно.

Потоки могут разделять экземпляры структур (они глобальные – Системные таблицы). Ядро и функции ядра довольно плохо описаны. В UNIX любой процесс создаётся системным вызовом fork (Есть в ядре функция clone). При этом создаётся новый процесс – процесс потомок, который находится в отношениях с процессом, вызвавшим процесс, как потомок к предку. Это отношение не математическое, так просто говорят. Это отношение поддерживается соответствующими указателями в системе. А именно процесс вызвавший новый процесс … потомок получает указатель на своего предка. В Linux процесс описывается структурой, UNIX – struct proc{…} (выдадут в методичке), в Linux – struct task-struct{…} (Найти в интернете, распечатать, несколько страниц, называется дескрипторами процесса, большинство полей – указатели на другие структуры, и эта структура содержит всю необходимую информацию чтобы система могла управлять процессом и выделять ему ресурсы). Естественно, это структура ядра, они не доступны в User’е.

Как мы уже сказали, после fork создается новый процесс – процесс-потомок, который является копией процесса предка в том смысле, что потомок наследует код предка, дескрипторы открытых файлов, сигнальную маску, окружение. В старых системах код предка копировался в адресное пространство потомка, то есть – потомку создавалось собственное адресное пространство. (Что значит создать защищённое виртуальное пространство? – надо создать соответствующие таблицы – в старом UNIX – таблицы сегментов, в новой системе – таблицы страниц). Таблицы страниц – образуют адресное пространство процесса. Очевидно, что это крайне неэффективно, то есть в системе могут одновременно существовать какое-то кол-во копий одной и той же программы. Поэтому в современных системах существует два способа оптимизации задачи создания нового процесса. (можно встретить такое название - оптимизация fork)

**Первый способ** называется копирование при записи (copy on write). Когда вызывается системный вызов fork и создаётся новый процесс, для него создаются (карты таблицы адресов), таблицы страниц и они ссылаются на страницы адресного пространства предка, при этом для страниц данных из стека предка, права меняются на only read и устанавливается флаг copy on write. Если предок или потомок пытаются изменить какую-либо страницу, возникает исключение по правам доступа. Обрабатывая это исключение, система обнаружит установленный флаг copy on write и создаст копию нужной страницы в адресном пространстве того процесса, который пытался её изменить. Дескриптор этой страницы должен быть добавлен в таблицу страниц этого процесса – сначала дескриптор, потом всё остальное. ОС – программа, не может работать … . В результате будут созданы копии только нужных страниц. Как только было найдено copy on write решило проблему коллективного … и страничное преобразование стало ….

?Такая ситуация с изм правами доступа к? .. и установлен флаг copy on write существует пока ?...потомок? не вызовет или системный вызов exit() или системный вызов exec(). Exit() – системный вызов завершения процесса.

**Второй способ.** Реализован способ с системным вызовом vfork(). В этом случае для потомка не создаются собственные карты трансляции адресов, а предок предоставляет потомку свои, при этом предок блокируется до того момента пока потомок не вызовет exit или exec.

В результате fork создаётся иерархия процессов. Эта иерархия поддерживается указателями, в UNIX есть дерево, но в UNIX есть двусвязные списки. (Таблица — это массив структур, проблема массивов в доп. накладных расходах при удалении и вставке) (Двусвязный список – для ускорения поиска. Сортировки для использования специальных методов поиска)

Процессы в списке находятся в соответствии с приоритетом.

В системе всегда есть процесс с идентификатором 0 и 1. Все идентификаторы в UNIX – это целые положительные числа. Процесс 0 – это процесс, запустивший систему. Процесс 1 – это процесс открывший терминал. (Терминал – это клавиатура и дисплей. Какая идеология ОС UNIX? – разделение времени (sharing time)).

В системе может быть создано большое кол-во терминалов и каждого будет процесс 1. Процесс 1 – предок всех процессов запущенных на данном терминале.

**Рисунок дерева**

Процесс у которого завершается предок – процесс сирота. При завершении процесса система проверяет не остались ли у этого процесса не завершившиеся потомки, если да, то запускается усыновление. Этого сироту усыновит процесс 1. При этом предок получит указатель на нового потомка, а предо указатель на нового предка. Fork на русском значит - вилка.  
#include <stdio.h>  
# include <unistd.h>  
int main(void)  
{  
 int childpid;  
 if ((childpid =fork())==-1)  
 {  
 perror(“Can’t fork”)  
 exit(1)  
 }  
 else if (childpid == 0)  
 {  
 exec(); //условная запись  
 printf(“Child;pid=%d,ppid=%d\n”,getpid(),getppid());  
 return 0; // Возвращает управление предку  
 }   
 else  
 {  
 wait(&status)  
 print(“parent: childpid =%d, pid=%d\n”, childpid, getpid());  
 }  
 return 0;  
}

Parent получает идентификатор потомка  
Соглашение UNIX – если завершается с ошибкой, возвращается -1.  
Для того чтобы не появлялись сироту в UNIX есть системный вызов wait(&status). Система не контролирует, где мы вызвали wait, в предке или в потомке (Любой процесс может стать предком). Также система не контролирует, где мы вызываем exec. Это дополнительные проверки, которые системы не интересны. Правильная логика, следующая: предок должен ждать завершения своих потомков. То есть в предке надо писать wait(&status) – вернёт статус завершения потомка.

Но с wait’ом связана проблема.

**Рисунок 2 (вечный wait)**

Это решается с помощью состояние зомби. В UNIX все процессы проходят через состояние зомби. Зомби — это процесс у которого отобраны все ресурсы кроме строки в таблице процессов (дескриптор).

### Системный вызов Exec().

(Нужно выучить про fork и exec?)

Exec создаёт низкоуровневый процесс (На самом деле процесс не создаётся, процесс создаётся fork’ом). Для программы, которая передаётся exec в качестве параметра, создаётся адресное пространство, то есть таблица страниц. Exec проверяет права доступа этого процесса (потомка) к файлу, проверяется путь к файлу и является ли он исполняемым. Три типа файлов в системе: исполняемый, объектный, исходник. Exec может быть передан только исполняемый файл. После этого создаётся адресное пространство, но у child’а уже оно есть, надо убрать (уничтожить) эти страницы. Очевидно, что в дескрипторе процесса должна быть строка, содержащая адрес таблиц страниц, и надо поменять его, указав адрес новых таблиц страниц. Передать управление на точку вход (передать адрес точки входа в instruction pointer). То есть системный вызов exec() переводит процесс на новое адресное пространство.

Fork-бомба (почему называется fork бомба? :D)

Любая программа в два этапа:

1. Создаётся процесс
2. Переход в новое адресное пространство

У exec() есть несколько вариантов.

execl(“/bin/ls”, “ls”, “-l”, 0);  
execl(char \*name, char arg[0], …, char arg[n], 0);