

Разанова Наталья Юрьевна
Курс „Операционные системы“

Книга Стивенса и Раго „UNIX. Профессиональное программирование“;

Оттуда берём материал по демонам и файлам;
Материалы по модулям ядра: Цирюльник,
IBM.com;

Защищённый репсизм-32-разр.;
„весь репсизм юнд тебе ленист на защищён-
ном репсизме“;

Процессы UNIX

Историю надо знать, тогда всё становится понятным;

Полноценный UNIX появился в 70-е для PDP-11;

Люди придумали язык Си для себя;

Все макроязыки основаны на Си;

Windows изучать невозможно, т.к. у них закры-
тые исходные коды;

Linux - UNIX-подобная ОС;

⇒ Linux построен на базовых понятиях
(парадигмах) UNIX;

- Открытый код;

Минус Порвальдс

„В Linux есть всё“: все сист. вызовы UNIX BSD, System V, POSIX;
(Five);

Ф-ции ядра (в отличие от API) можно менять, т.к. они не стандартизованы POSIX;

Процесс — ^(главная) основная абстракция ОС;

Процесс — программа в стадии выполнения;

Выполняться может только исполняемый файл, т.е. программа, прошедшая компоновку и linking;

exe — виндовое понятие;

В UNIX понятие „процесс“ явл. кардинальным;

Процесс в UNIX часть времени выполняется в режиме пользователя (тогда выполняется его код), а часть — в режиме ядра (тогда выполняется неиспользуемый код ядра ОС);

Всё рассматривается относительно процесса (UNIX „заточен“ на процессы);

Любой процесс можно создать, используя сист. вызов **fork()** („вышка“, но на самом деле это „развышка“ перед 2 дорогами);

Сист. вызов **fork()** создаёт новый процесс — процесс-потомок, который явл. копией процесса-предка, наследуя код предка, дескрипторы открытых файлов, маску создания файлов, сигналь-

ную маску и т.п.;

В старых версиях UNIX код предка копировался в адр. пр-во потомка, т.е. для потомка создавалось свое вирт. адр. пр-во, в которое копировался код предка \rightarrow в памяти могло лежать копий кода - не оптимально;

В совр. системах используется "оптимизированный" Fork();

Система операционная, т.к. заменяет оператора

\hookrightarrow самое важное слово

Состоит из связанных между собой подсистем (упр-е памятью, внеш. устр-вами, процессорами и т.д.)

Но это всё действует как единая система, из которой нельзя ничего выкинуть;

UNIX начали писать с файловой подсистемы
(есть во всех ОС)

Что значит создать вирт. адр. пр-во?

\hookrightarrow абстракция системы;

ОС большую часть времени оперирует вирт. адресами;

Вирт. адр. пр-во необходимо описывать;

Старое название - карты трансляции адресов;

Новое - таблицы страниц;

Для нового процесса создаются таблицы страниц;

В программировании любая таблица - массив структур;

Основная таблица в ОС - таблица процессов;
На самом деле её нет, а ОС оперирует связанными списками (чаще всего двусвязными);

С таблицами очень неудобно работать, т.к. они непереворотливые;

Оптимизация Fork()

Flag copy on write;

Создается копия процесса - предка, собств. адр. пр-во (собств. таблицы страниц), а дескрипторы страниц потомка (describe - описывающая структура) ссылаются на страницы адр. пр-ва предка (в struct есть указатель на адр. пр-во)

Для страниц адр. пр-ва предка обычные права доступа read/write (rw) меняются на only read (r) и устанавливается флаг copy on write (копирование при записи);

Если предок/потомок попытается изменить какую-либо страницу, возникнет исключение прав доступа;

Обрабатывая его, ОС увидит уст. флаг copy on write и создаст копию данной страницы в адр. пр-ве того процесса, который пытался её изменить;

⇒ создаются только копии нужных страниц;

Семанты программы (ASM):

код, данные и стек,
(only read) (read write)

↳ а после fork() меняются
на only read;

В рез-те появления этого орлага все совр. ОС
базируются на вирт. страничной памяти
(упр-е памятью страницами по запросу);

Страницы удобно менять (механизм paging);

fork:

```
pid_t childpid;  
if ((childpid = fork()) == -1)  
{  
    /* ошибка fork */  
}  
else if (childpid == 0)  
{  
    /* потомок */  
}  
else  
{  
    /* предок */  
}
```

Сообщение UNIX: ошибка сист. вызова возв-
ращает (-1);

Процесс может создать любое кол-во потомков;

Форк - Бамба - ОС теряет работоспособность, т.к. любой процесс (даже только что созданный) потребляет ресурсы;

В рез-те вызова `fork()` в системе создается иерархия процессов в отношении „предок - потомок“;

ОС, поддерживающая его, имеет указатели в дескрипторе процесса;
(структура)

→ В Linux: `struct task_struct`;

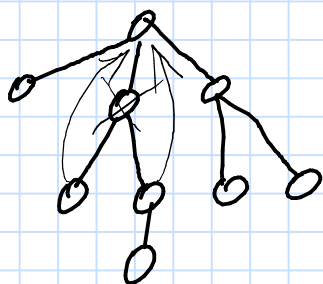
В UNIX BSD: `struct proc`;

Любой процесс описывается дескриптором (структурой), содержащим указатель на процесс-предок;

Что происходит при завершении родителей?

В UNIX есть процессы - сироты;

Чтобы не рушить иерархию, этих потанков усыновляет процесс, открывший терминал ($pid=1$);



Процессы между проц. с $pid=1$ и проц., запущенными на данном термине, (сироты) переходят на него;

В системе всегда есть процессы с $pid=0$ („init“, запустивший систему) и $pid=1$ (процесс, открывший/запустивший терминал);

→ В любом терминале это так (т.к. их можно

открыть много, т.к. они виртуальные);

⚡ Чтобы сироты не возникали, есть сист. вызов `wait (& status)`

↳ статус завершения потомка;
↳ предок ждет завершения потомков;

Итого: параллельно выполняются 2 копии одной программы (предок и потомок);

Такая ситуация с правами доступа к адр. пр-ву и установлением флага `core on write` будет существовать в системе до тех пор, пока потомок не вызовет сист. вызов `exit()` (завершение процесса) или сист. вызов `exec()`, который переводит процесс-потомок на адр. пр-во программы, которая передана `exec()` в качестве параметра;

В рез-те потомок начинает выполнять другую программу;

В UNIX, чтобы запустить программу, надо сначала вызвать `fork()`, а затем процесс-потомок должен вызвать `exec()`;

Запуск программы:

`fork` → процесс → `exec` → программа;

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

```
int main(void)
```

```
{
```

```
    pid_t childpid;
```

```
    if(( childpid = fork()) == -1)
```

```
    {
```

```
        perror("Can't fork");
```

```
        exit(1);
```

```
    }
```

```
    else if (childpid == 0) (можно не писать)
```

```
    {
```

```
        printf("child: pid = %d, ppid = %d, gid = %d\n",
               getpid(), getppid(), getgid());
```

```
    }
```

```
    else
```

```
    {
```

```
        printf("parent: pid = %d, childpid = %d, gid = %d\n",
               getpid(), childpid, getgid());
```

```
    }
```

```
    return 0;
```

```
}
```

Для завершения надо вызвать `wait()`, но мы хотим пронаблюдать механизмы;

Процессы одной группы могут получать (а могут и игнорировать) одни и те же сигналы;

Сигналы - важнейшее средство информирования процессов в системе;

Важнейшее событие - завершение процесса;

Мы работаем с системой разделения времени;