Сергей Юрьевич Шилов

Системное программирование для современных платформ

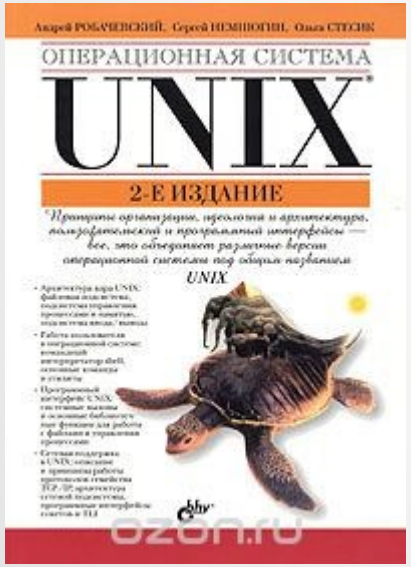
Системное программирование для современных платформ

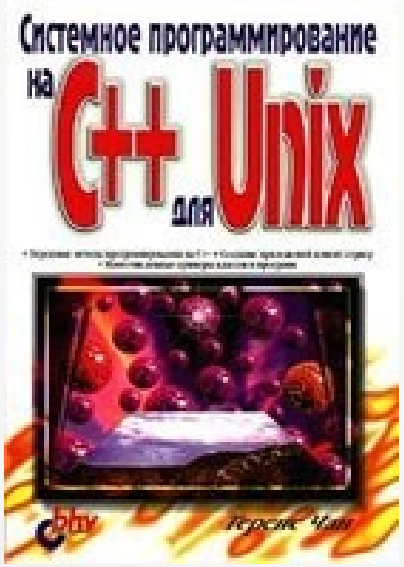
Содержание курса

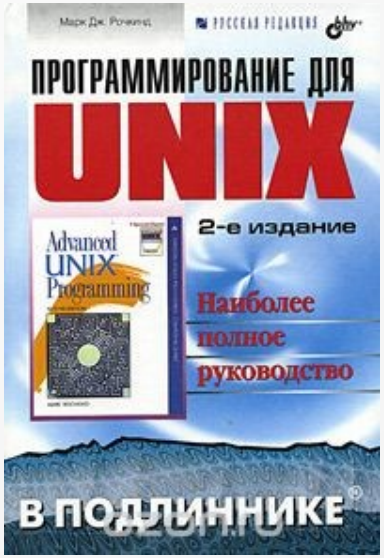
1. Введение. Архитектура ОС и стандарты программирования
2. Процессы. Атрибуты процессов, порождение процессов
3. Сигналы и таймеры
4. Низкоуровневый файловый интерфейс ввода-вывода
5. Взаимодействие процессов посредством функций ввода-вывода.
6. Сокеты
7. Файлы, отображаемые в память
8. InterProcess Communication
9. Фреймворк RPC
10. Multithreading programming

Системное программирование для современных платформ

Рекомендованная литертура







Системное программирование для современных платформ

Архитектура ОС и стандарты программирования

Системное программирование для современных платформ

Операционные системы:

- Windows (XP, Vista, 7, 8, 10, Server 2003, 2008, 2012)
- OpenSolaris
- HP-UX
- Linux (RedHut, Fedora, Gentoo, Debian, Ubuntu, Slackware, SUSE)
- FreeBSD, NetBSD, OpenBSD
- Mac OS X, OS X (FreeBSD-based)
- Android (Linux + Java-like VM)

Системное программирование для современных платформ

Операционные системы:

- Microsoft® Windows®
- UNIX™ and Unix-like

Системное программирование для современных платформ

Особенности современных операционных систем

- Вытесняющая защищённая многозадачность
- Поддержка аппаратной защиты
- Плоское адресное пространство для процессов
- Универсальный интерфейс ввода-вывода
- Поддержка структурированных файловых систем
- Контроль полномочий на основе списков контроля доступа (ACL)
- Синхронизация и взаимодействие между процессами
- Масштабируемость
- Совместимость снизу вверх
- Архитектуры x86, x64, IA-64, MIPS
- Форматы исполняемых файлов COFF, PE, ELF

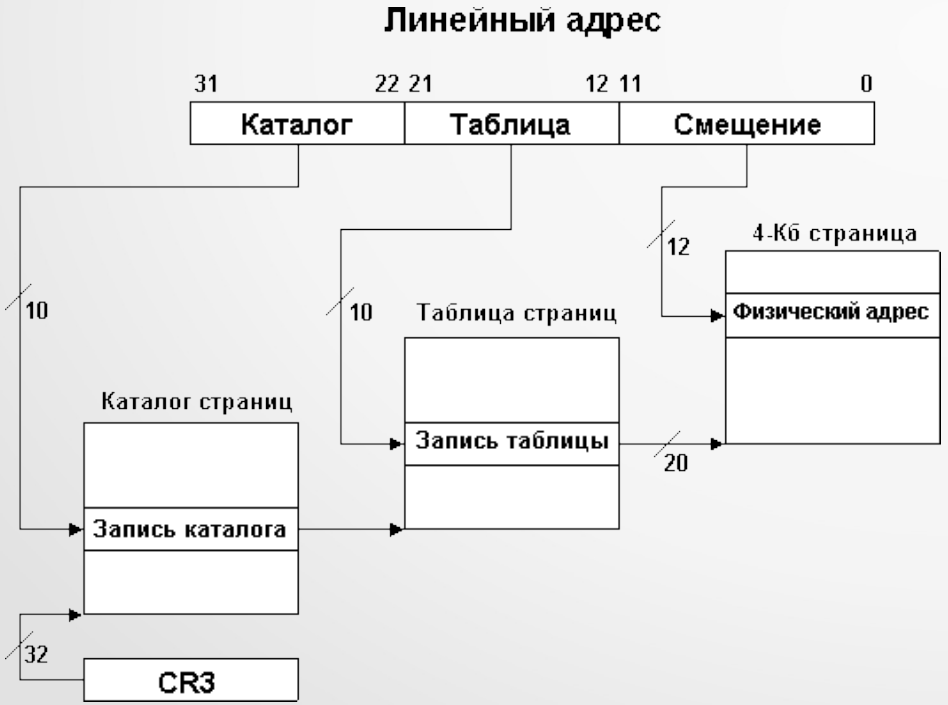
Системное программирование для современных платформ

Современная архитектура x86 и x64

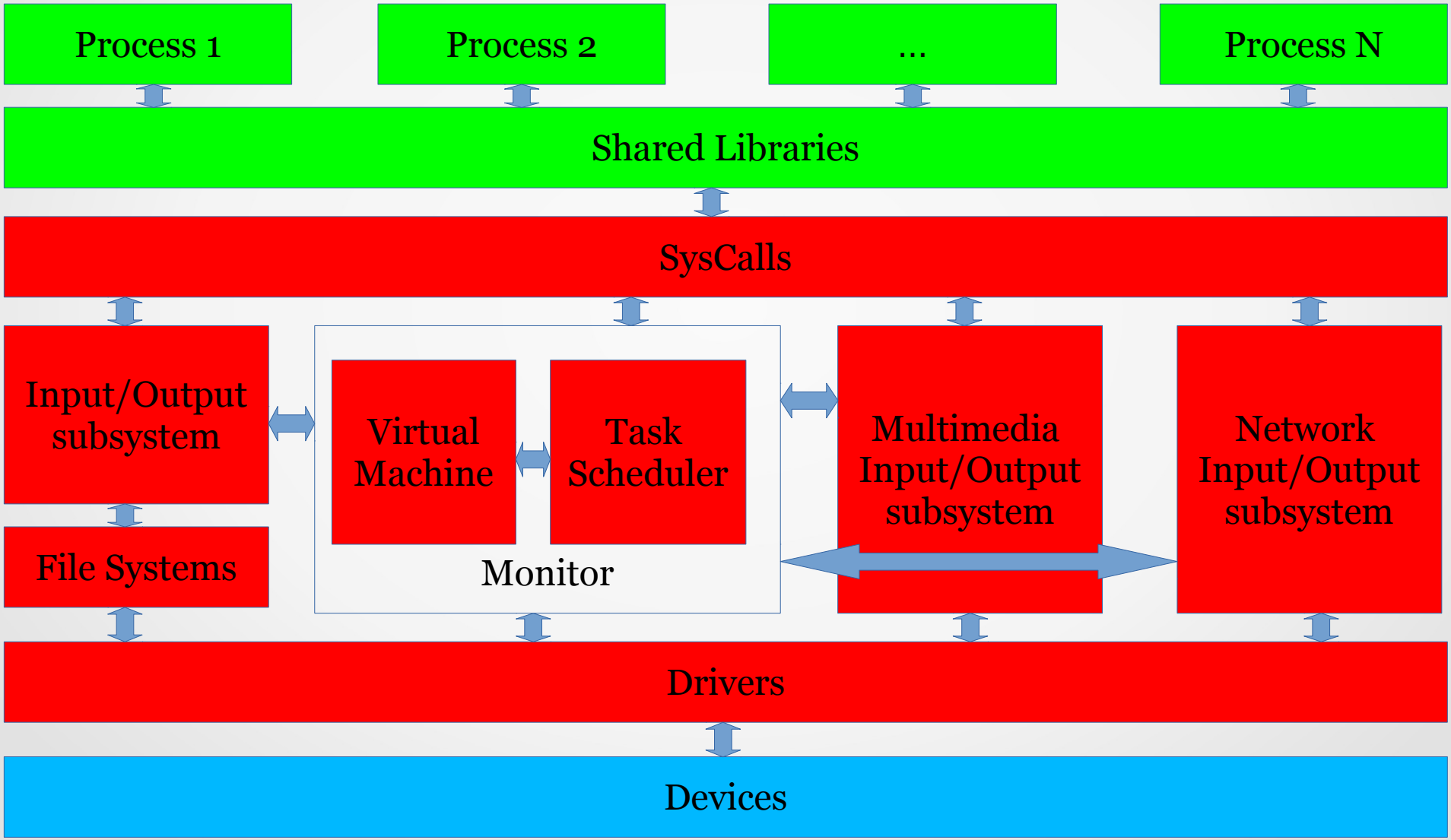
- Арифметические регистры EAX–EDX(32bit), RAX–RDX, RBP, RSI, RDI, RSP, R8–R15 (64bit)
- Адресные регистры CS, DS, ES, SS, FS, GS
- Регистры флагов
- FPU (x87) – extended IEEE-754
- 32 bit modes: Real, Protected, Virtual (V8086)
- 64 bit modes: Long, Legacy
- Page size: 4K, 4M (PAE), 1G (64bit)
- Protection: Ring 0 – Ring 3
- Система команд x86, x64 эмулируются микрокодом

Системное программирование для современных платформ

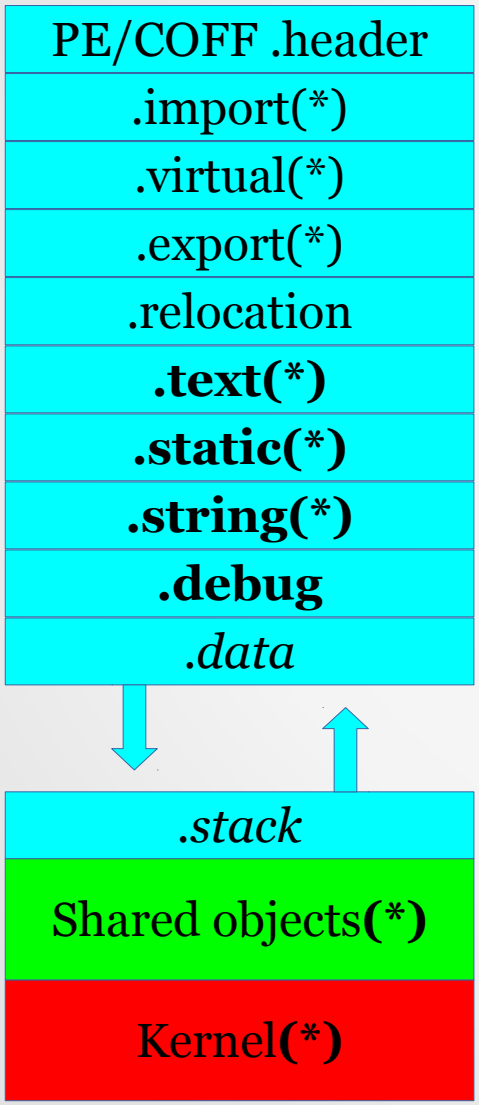
Страничная адресация: механизм преобразования виртуальных (логических) адресов в физические адреса. На уровне ОС это преобразование выполняется виртуальной машиной, на аппаратном – при помощи MMU. У ранних процессоров x86 роль MMU выполняло ALU.

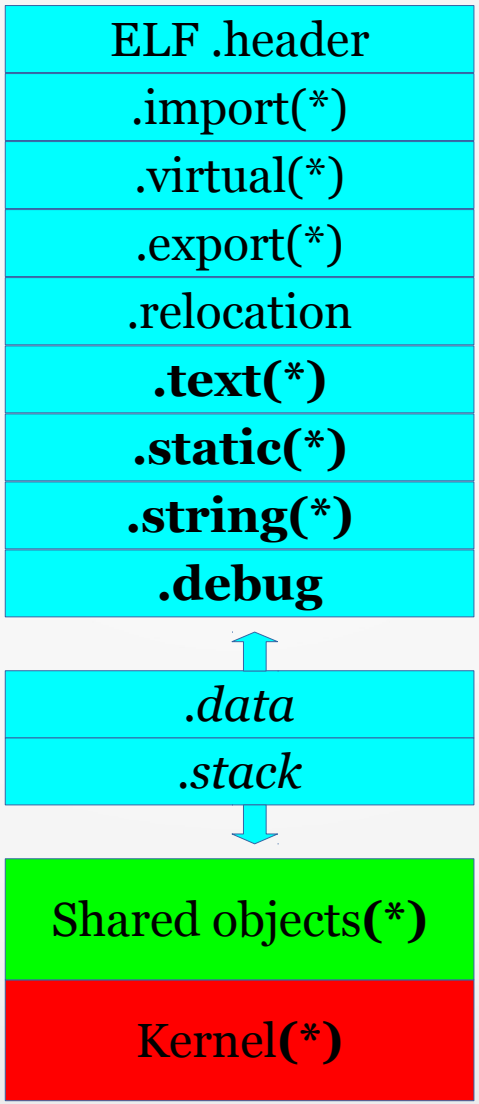


Системное программирование для современных платформ



Системное программирование для современных платформ





Legend:
.text – precompiled
.data – dynamic
(*) – shared

Системное программирование для современных платформ

Стандарты программирования:

- ANSI C (C89)
- Spec 1170
- Single UNIX Specification: System interfaces and headers
- POSIX 1, 1b, 1c

Системное программирование для современных платформ

Атрибуты процессов.
Системные вызовы fork() и семейство exec

Системное программирование для современных платформ

Атрибуты процессов

Идентификатор процесса (**PID**) – уникальный номер процесса, индекс процесса в таблице task scheduler'a и других системных таблицах

Идентификатор родительского процесса (**PPID**)

Поправка приоритета (**NI**) – относительный приоритет процесса, учитываемый планировщиком при определении очередности запуска. Может быть изменён пользователем при помощи утилиты **nice**.

PRI – фактический приоритет выполнения, зависящий от нескольких факторов, в частности от заданного относительного приоритета.

TTY – управляющий терминал процесса. Терминал или псевдотерминал, связанный с процессом. С этим терминалом по умолчанию связаны стандартные потоки: входной, выходной и поток сообщений об ошибках. Может отсутствовать (процессы-демоны, daemons)

Системное программирование для современных платформ

Атрибуты процессов (продолжение)

Реальный (**UID**) и эффективный (**EUID**) идентификаторы пользователя
UID процесса является ID пользователя, запустившего процесс. **EUID** служит для определения прав доступа процесса к системным ресурсам (в первую очередь к ресурсам файловой системы). Обычно реальный и эффективный идентификаторы совпадают, т.е. процесс имеет в системе те же права, что и пользователь, запустивший его. Однако существует возможность задать процессу более широкие права, чем права пользователя, путём установки бита **SUID**, когда эффективному идентификатору присваивается значение идентификатора владельца выполняемого файла (например, пользователя root).

Реальный (**GID**) и эффективный (**EGID**) идентификаторы группы
GID процесса равен ID основной или текущей группы пользователя, запустившего процесс. **EGID** служит для определения прав доступа к системным ресурсам от имени группы. Обычно эффективный идентификатор группы совпадает с реальным. Если для выполняемого файла установлен бит **SGID**, такой файл выполняется с эффективным идентификатором группы-владельца исполняемого файла.

Системное программирование для современных платформ

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

ОПИСАНИЕ

fork создает процесс-потомок, который отличается от родительского только значениями PID (идентификатор процесса) и PPID (идентификатор родительского процесса), а также тем фактом, что счетчики использования ресурсов установлены в 0. Блокировки файлов и сигналы, ожидающие обработки, не наследуются.

В настоящее время **fork** реализован с помощью copy-on-write (COW), поэтому расходы на **fork** сводятся к копирования таблицы страниц родителя и созданию уникальной структуры, описывающей задачу.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При успешном завершении родителю возвращается PID процесса-потомка, а процессу-потомку возвращается 0. При ошибке родительскому процессу возвращается -1, процесс-потомок не создается, а значение errno устанавливается должным образом.

ОШИБКИ

EAGAIN

Превышено максимальное число процессов или размер таблиц ядра

ENOMEM

Не хватает памяти для создания нового процесса.

Системное программирование для современных платформ

EXAMPLE:

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

int main (void) {
   pid_t p;
   printf("Original program,  pid=%d\n", getpid());
   p = fork();
   if (p == 0) {
       printf("There is in child process, pid=%d,
                ppid=%d\n", getpid(),  getppid());}
   else {
    printf("There is in parent, pid=%d,
                fork returned=%d\n", getpid(), p);}
}
```

Системное программирование для современных платформ

```
#include <unistd.h>
extern char **envp;

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg , ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

ОПИСАНИЕ

Семейство функций **exec** заменяет текущий образ процесса новым образом процесса. Начальным параметром этих функций будет являться полное имя файла, который необходимо исполнить.

Функции **execlp** и **execvp** дублируют действия оболочки, относящиеся к поиску исполняемого файла, если указанное имя файла не содержит символ черты (/). Путь поиска определяется в окружении переменной **PATH** . Если эта переменная не определена, то используется путь поиска "**:/bin:/usr/bin**" по умолчанию. Дополнительно обрабатываются некоторые ошибки.

Если запрещен доступ к файлу (при попытке исполнения **execve** была возвращена ошибка **EACCES**), то эти функции будут продолжать поиск вдоль оставшегося пути. Если не найдено больше никаких файлов, то по возвращении они установят значение глобальной переменной errno равным **EACCES**.

Системное программирование для современных платформ

ОПИСАНИЕ (продолжение)
Если заголовок файла не распознается (при попытке выполнения функции execve была возвращена ошибка ENOEXEC), то эти функции запустят оболочку (shell) с полным именем файла в качестве первого параметра. (Если и эта попытка будет неудачна, то дальнейший поиск не производится.)

Параметр **const char *arg** подразумевают параметры **arg0, arg1, ..., argn**, являющиеся указателями на **NULL**-терминированные строки, которые являются списком параметров, доступных исполняемой программе. **Arg0** должен указывать на имя, ассоциированное с исполняемым файлом. Список параметров также должен терминироваться **NULL**.

Функции **execv** и **execvp** предоставляют процессу массив указателей на **NULL**-терминированные строки, которые являются списком параметров, доступных исполняемой программе. **Arg0** должен указывать на имя, ассоциированное с исполняемым файлом. Массив указателей также должен терминироваться **NULL**.

Функции **execle** и **execve** также определяет окружение исполняемого процесса, помещая после указателя **NULL**, заканчивающего список параметров или после указателя на массив, **argv** дополнительного параметра. Этот дополнительный параметр является массивом уазателей на **NULL**-терминированные строки и также должен быть **NULL**-терминированным. Другие функции извлекают окружение нового образа процесса из внешней переменной environ текущего процесса.

Системное программирование для современных платформ

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ
0 – в случае успешного завершения, в случае ошибки – -1 и глобальной переменной errno будет присвоен код соответствующей ошибки.

EXAMPLE:

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

int main (void) {

        /* Определить массив с завершающим нулем команды для запуска
         следующим за любым параметром */
        char *arg[] = { "/usr/bin/ls", "-l", "./", 0 };

        /* fork и exec в порожденном процессе */
        if (fork() == 0) {
                printf("In child process:\n");
                execv(arg[0], arg);
                printf("I will never be called\n");
        }
        printf("Execution continues in parent process\n");
}
```

Системное программирование для современных платформ

```
#include <sys/types.h>
#include <unistd.h>
pid_t vfork(void);
```

ОПИСАНИЕ
Функция **vfork()** аналогична **fork()** за тем исключением, что поведение процесса, созданного **vfork()**, неопределено, если он модифицирует любые данные, кроме переменной типа **pid_t**, используемой в качестве значения, возвращаемого **vfork()**, или возвращается из функции, в которой была вызвана функция **vfork()**, или вызывает любую функцию до удачного исполнения **_exit()** или одной из функций семейства **exec**.

ОШИБКИ
EAGAIN
Превышено максимальное число процессов или размер таблиц ядра
ENOMEM
Не хватает памяти для создания нового процесса.

Системное программирование для современных платформ

Атрибуты	*fork ()*	*execl(), execv(), execlp(), execvp()*	*execle(), execve()*
precompiled exe sections	+	-	
dynamic exe sections	COW	-	
PID	-	+	
PPID	-	+	
UID, GID	+	+	
EUID, EGID	+	-	
Маска и обработчики сигналов	+	OS dependent	
Текущий и корневой каталоги	+	+	
Environment	+	+	-
Open files	+	+*	
Filemask	+	+	
TTY	+	+	
	*) excluse files with close-on-exec attribute		

Системное программирование для современных платформ

А что в Windows?

```
BOOL CreateProcess(LPCTSTR pNameModule, LPCTSTR pCommandLine,
SECURITY_ATTRIBUTES *pProcessAttr,
SECURITY_ATTRIBUTES *pThreadAttr, BOOL InheritFlag,
DWORD CreateFlag, LPVOID penv, LPCTSTR pCurrDir,
STARTUPINFO *pstartinfo,
PROCESS_INFORMATION *pProcInfo)
```

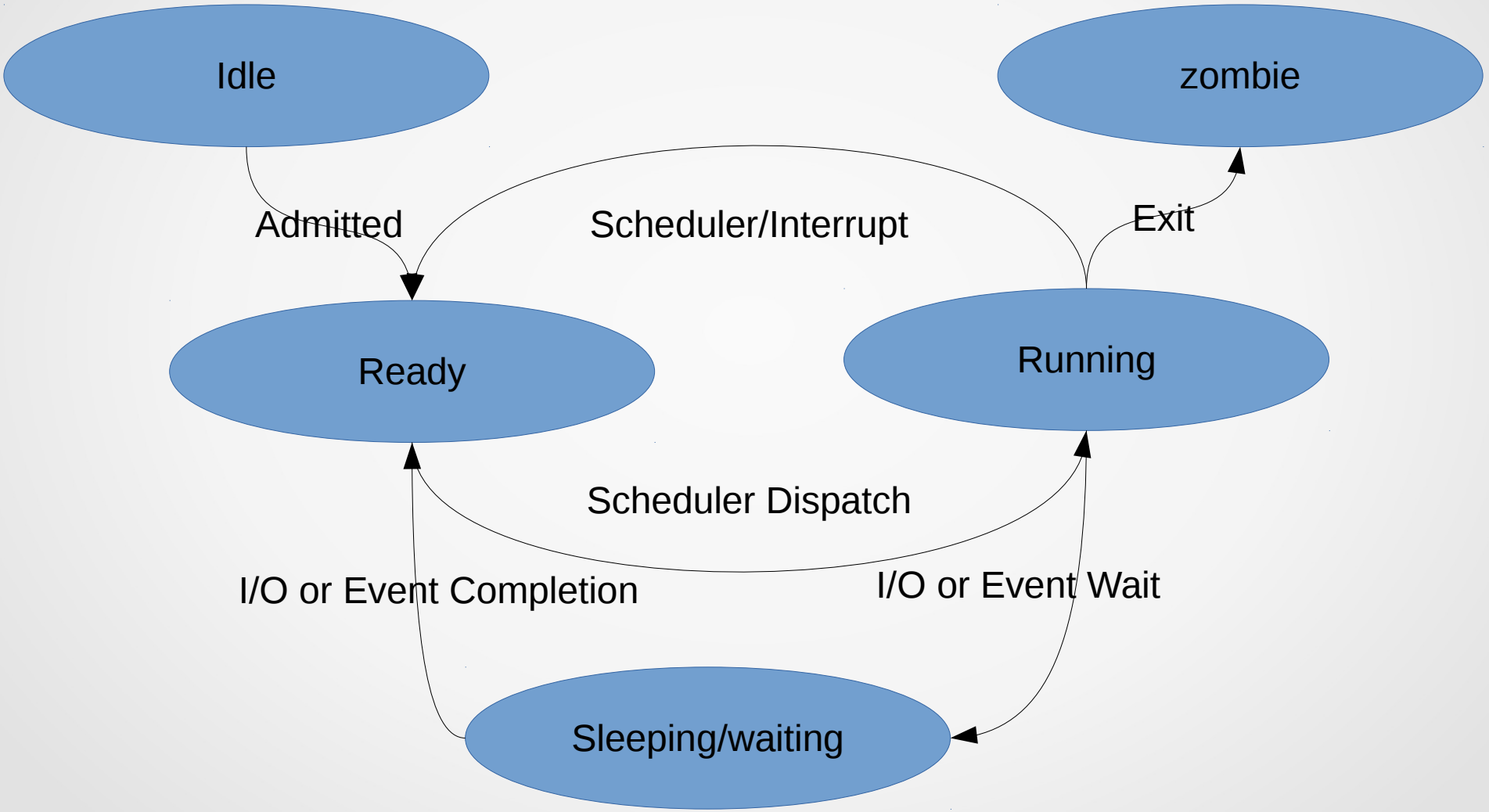
pNameModule – указатель на имя файла запускаемой программы
pCommandLine – указатель на текст командной строки
Поиск исполняемого файла осуществляется в том же каталоге, где находится EXE-файл родительского процесса. При не обнаружении требуемого файла, поиск продолжается в текущем каталоге вызывающего процесса, потом в системном каталоге Windows, затем в основном каталоге Windows и, наконец, в последовательности каталогов, перечисленных в переменной окружения PATH. Может не содержать расширения EXE. Параметр pNameModule позволяет задавать полное имя запускаемой программы или ее имя в текущем каталоге, причем обязательно с явно присутствующим в тексте расширением EXE.
pProcessAttr и **pThreadAttr** относятся к средствам расширенной защиты в Windows NT и в большинстве приложений не используются

Системное программирование для современных платформ

```
BOOL CreateProcess(LPCTSTR pNameModule, LPCTSTR pCommandLine,
SECURITY_ATTRIBUTES *pProcessAttr,
SECURITY_ATTRIBUTES *pThreadAttr, BOOL InheritFlag,
DWORD CreateFlag, LPVOID penv, LPCTSTR pCurrDir,
STARTUPINFO *pstartinfo,
PROCESS_INFORMATION *pProcInfo)
```

InheritFlag управляет наследованием объектов дочерним процессом
pCurrDir задает новый текущий каталог для запускаемого процесса
penv задаёт окружение для дочернего процесса
CreateFlag задает флаги создания процесса (обычно NORMAL_PRIORITY_CLASS)
Структура **pstartinfo** соответствует структуре описания сессии и содержит 18 полей, определяющих в первую очередь характеристики окна для создаваемого процесса. В простейших случаях все поля могут быть заполнены нулями, кроме поля с именем cb, в котором должна быть записана длина экземпляра этой структуры в байтах.
Структура **pProcInfo**, задаваемая адресом в последнем параметре, содержит четыре поля для возврата учетной информации из ОС после создания нового процесса.

Системное программирование для современных платформ



Системное программирование для современных платформ

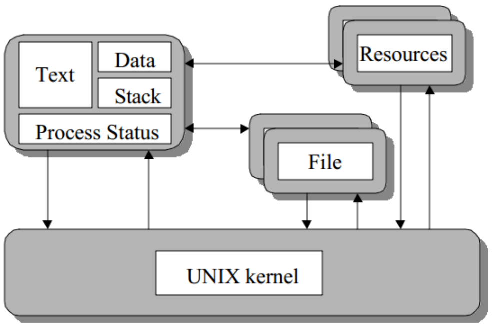
Спасибо за внимание!

:-)

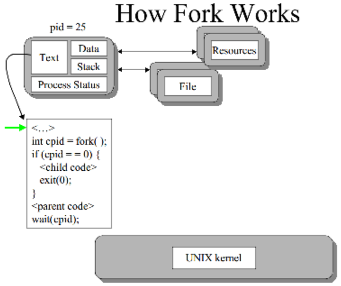
Системное программирование для современных платформ

Резервные слайды

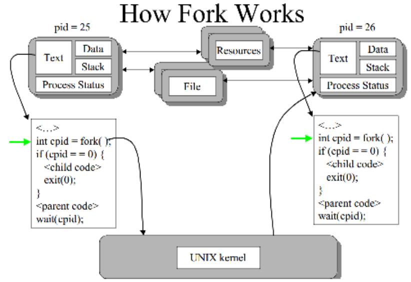
A Unix Process



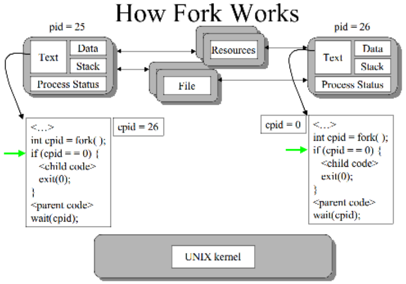
How Fork Works



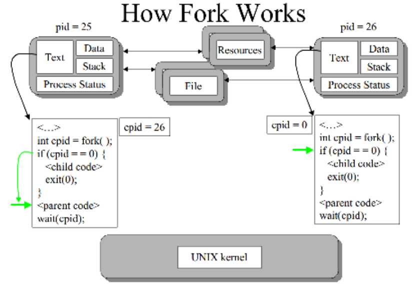
How Fork Works



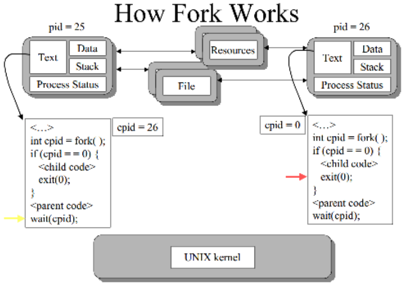
How Fork Works



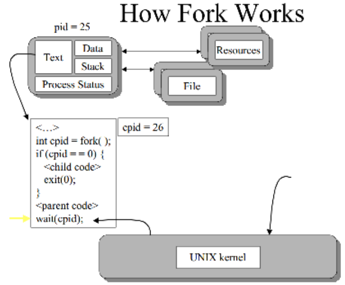
How Fork Works



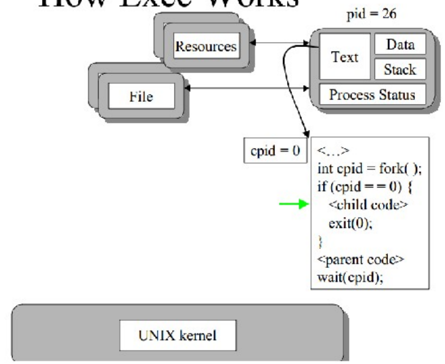
How Fork Works



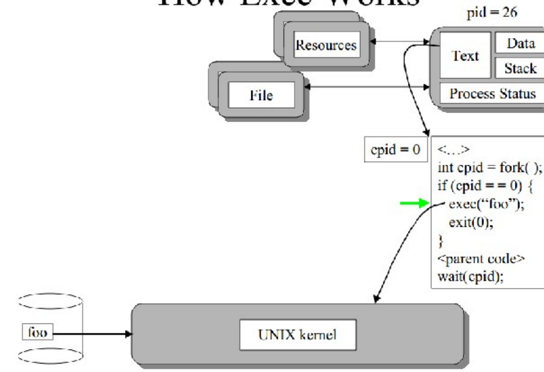
How Fork Works



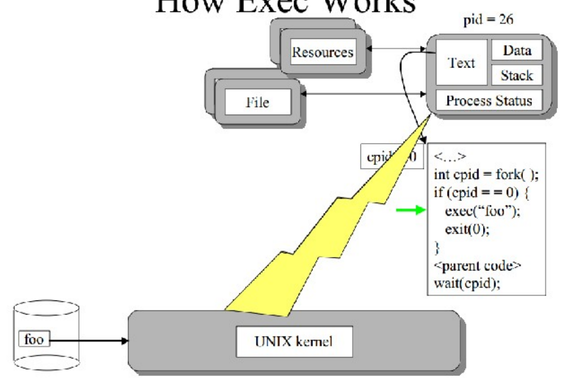
How Exec Works



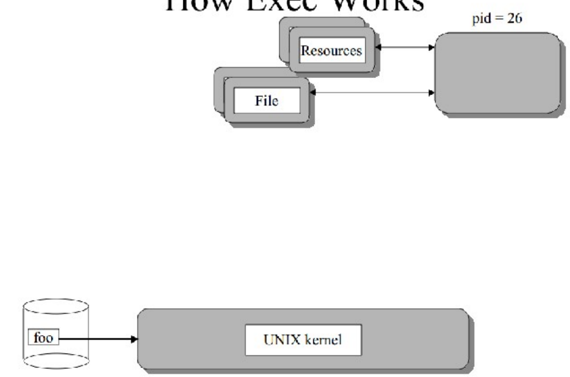
How Exec Works



How Exec Works



How Exec Works



How Exec Works