Понятие процесса

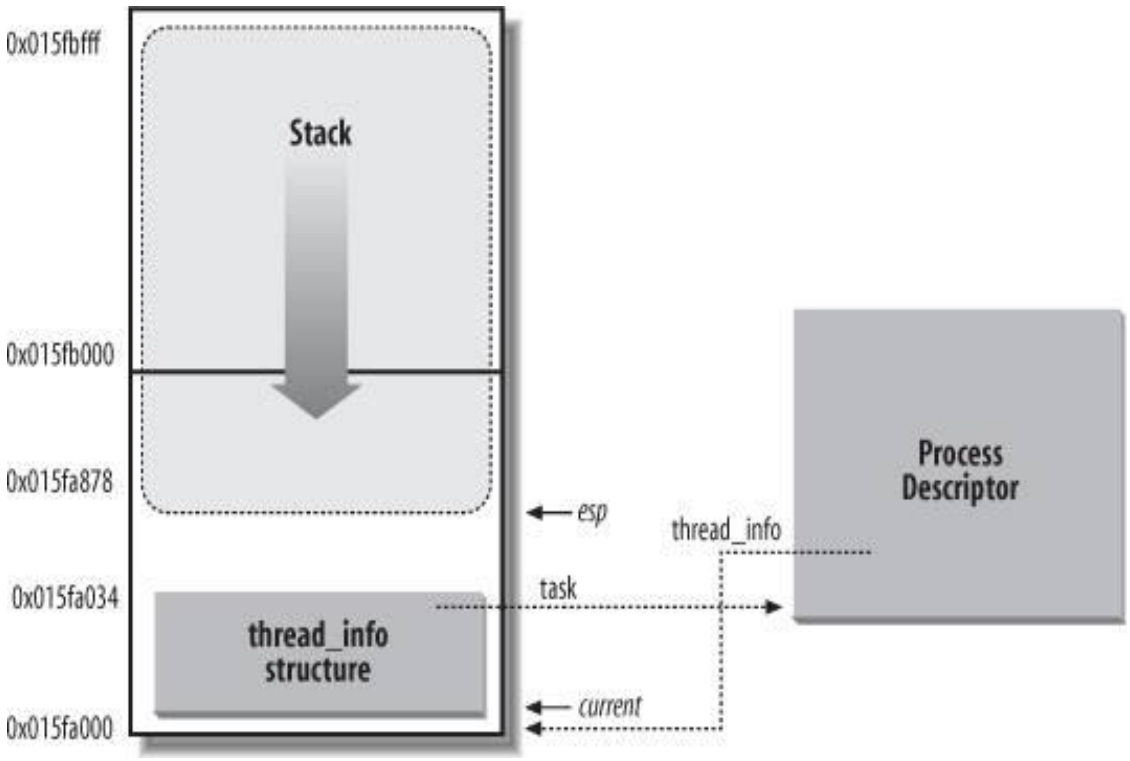
Процессом или задачей (Task) называют следующую совокупность объектов:

- программу, которую должен выполнить процесс,
- ресурсы, необходимые для ее выполнения,
- дескриптор, хранящий характеристики и данные процесса и представляющий процесс в ОС.

T = < P, R, D >

Размещение дескриптора процесса

Стек ядра задачи

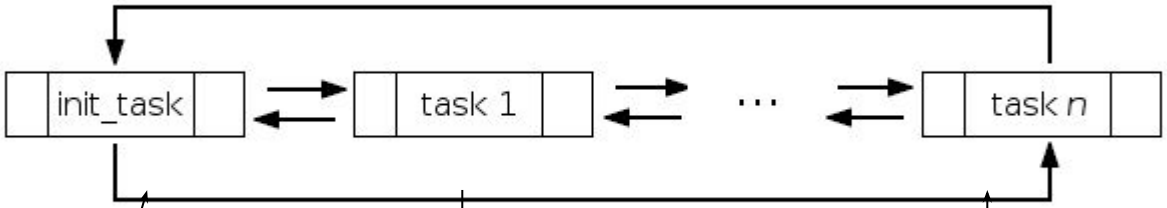



```
struct thread_info {
struct task_struct *task;
struct exec_domain *exec_domain;
unsigned long flags;
__u32  cpu; /*номер процессора*/
__s32 preempt_count;
__u32 rar_saved;
__u32 rsr_saved;
struct restart_block restart_block;
__u8 supervisor_stack[0];
};
```

Фрагмент описания дескриптора процесса

```
struct task_struct {
    unsigned long       state;   состояние
    void *stack;    указатель на стек ядра
    int prio, static_prio, normal_prio;   приоритет
    unsigned long       policy; /*     */ политика планирования
    struct task_struct   *parent; /*       */ указатель на родительскую задачу
    pid_t               pid;    идентификатор процесса
    pid_t               tgid;    идентификатор группы потоков
    int                 exit_state   код завершения
    struct mm_struct     *mm;    указатель на адресное пространство
процесса
    struct fs_struct     *fs;     информация о файловой системе
    struct files_struct  *files;  информация об открытых файлах
    struct thread_struct  thread;  данные процесса
    struct list_head     tasks;   список всех процессов
    struct signal_struct *signal;  обработчик сигналов
    char comm[TASK_COMM_LEN];  имя исполняемого файла
    ...
}
```

Список и хеш-таблица процессов



PID

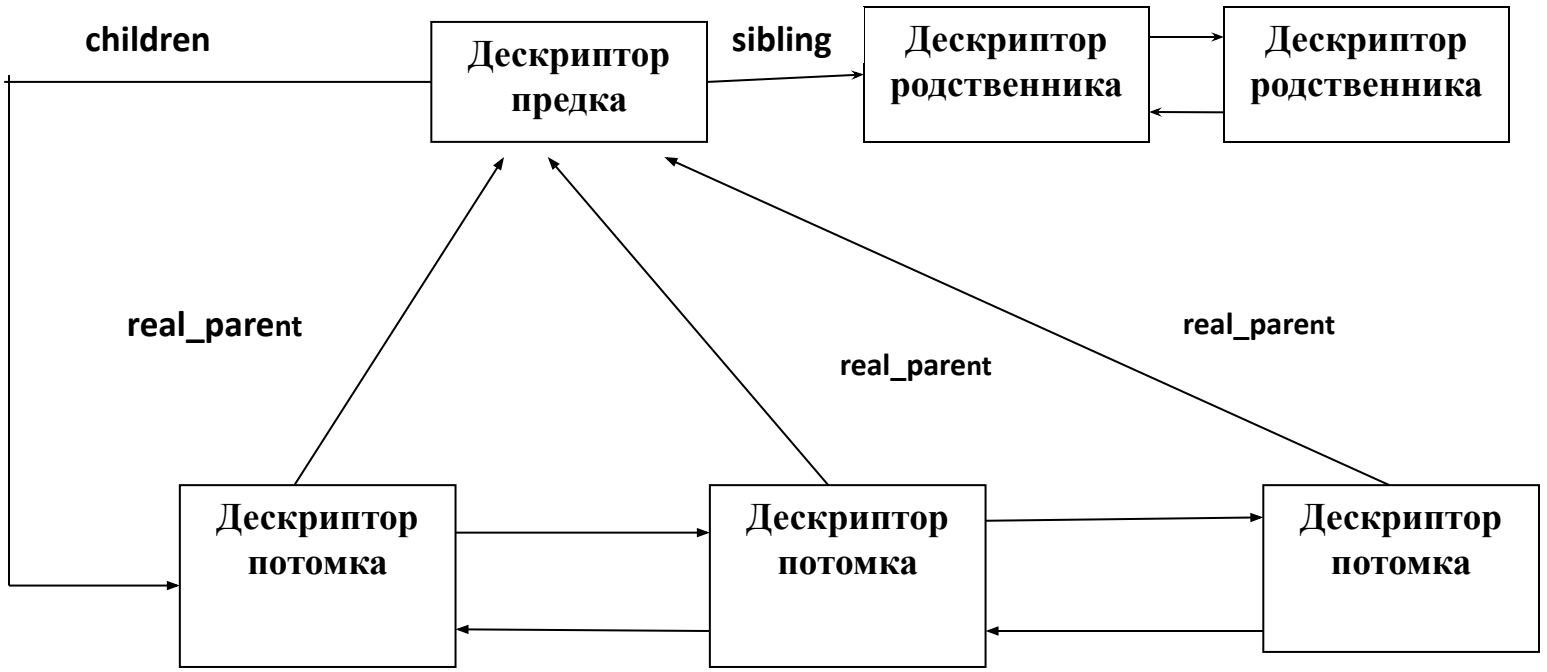
Хеш-таблица

Дескрипторы процессов

LXC-контейнер (Это система виртуализация уровня ОС, которая позволяет внутри хост операционной системы запустить дочерние ОС, у каждой из которых: своя виртуальная память, своя файловая система , свой сетевой стек. При этом ядро остаётся от операционной системы хоста.)

Пространство имен PID
Изначально все процессы в системе используют общие пространства имен.

Дерево процессов



Правила, по которым живут процессы

1. Каждому процессу в операционной системе соответствует запись в списке процессов.
2. Иерархия процессов в Linux имеет древовидную структуру.
3. У каждого процесса есть уникальный идентификатор PID.
4. У каждого процесса есть идентификатор группы потоков tgid. Группе присваивается идентификатор, соответствующий идентификатору первого процесса многопоточного приложения.
5. Если процесс умирает, то все его дочерние процессы передаются предку этого процесса , а сам процесс переходит в состояние “зомби”.
6. Родитель может уничтожить любого из дочерних процессов.

Учетные записи пользователей

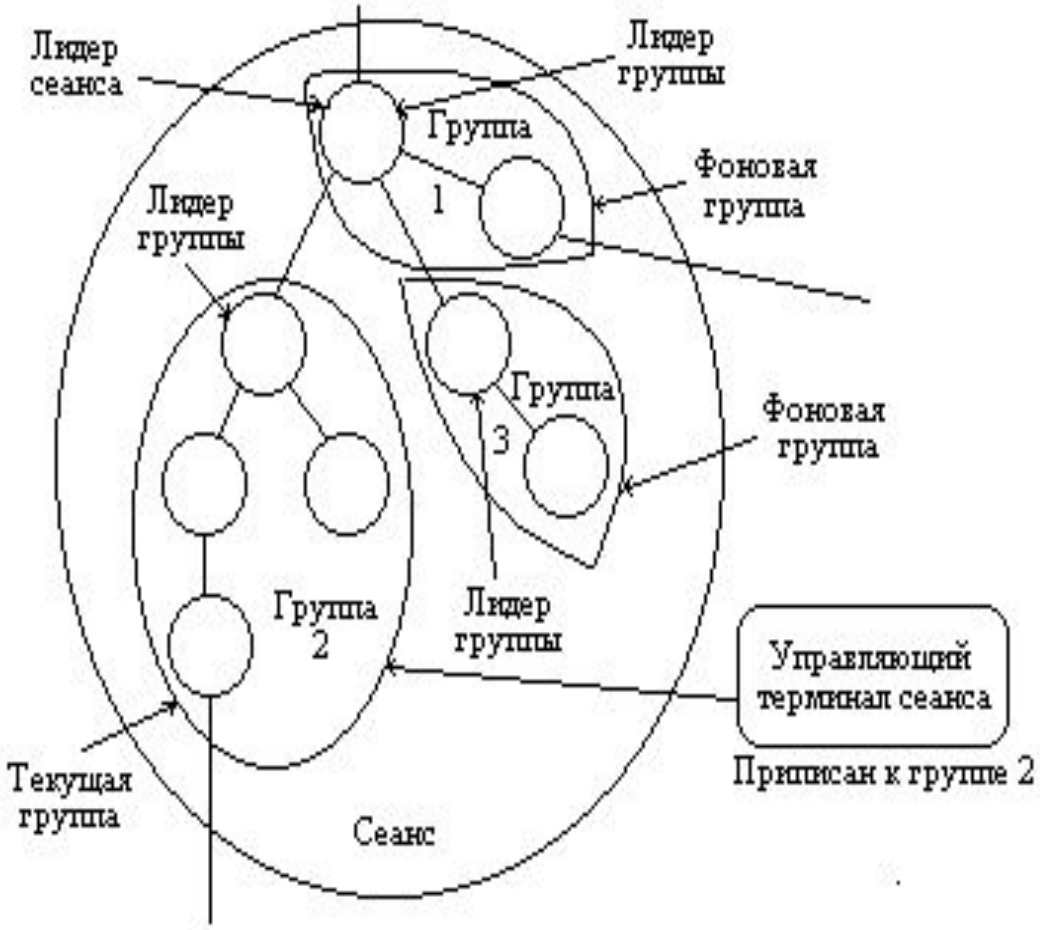
Учётная запись - Объект системы, при помощи которого Linux ведёт учёт работы пользователя в системе.

Входное имя - Название учётной записи пользователя, которое нужно вводить при регистрации пользователя в системе.

Идентификатор пользователя (UID)- Уникальное число, однозначно идентифицирующее **учётную запись** пользователя. Таким числом снабжены все **процессы** и все объекты **файловой системы**. Используется для персонального учёта действий пользователя и определения **прав доступа** к другим объектам системы.

Групповой идентификатор пользователя (**GID**) - Уникальное число, однозначно идентифицирующее группу пользователей. **Группы пользователей** применяются для организации доступа нескольких пользователей к некоторым ресурсам.

Сеансы и группы процессов



Сеанс - все процессы, запущенные пользователем.
Идентификатор сеанса sid – это pid первого запущенного процесса (**лидер сеанса**), обычно командная оболочка.
Терминал, к которому относится сеанс (ввод-вывод), называется **управляющим терминалом.**
Группа – процессы, запущенные одной командой.
Идентификатор группы pgid – это pid первого процесса группы (лидер группы)

Функции чтения атрибутов процесса

#include <sys/types.h>

#include <unistd.h>

pid_t getpid(void); идентификатор процесса

pid_t getppid(void); идентификатор предка

pid_t getsid(pid_t pid); идентификатор сеанса процесса

pid_t getpgid(pid_t pid); идентификатор группы процессов

uid_t getuid(void); реальный идентификатор пользователя

uid_t geteuid(void); эффективный идентификатор пользователя

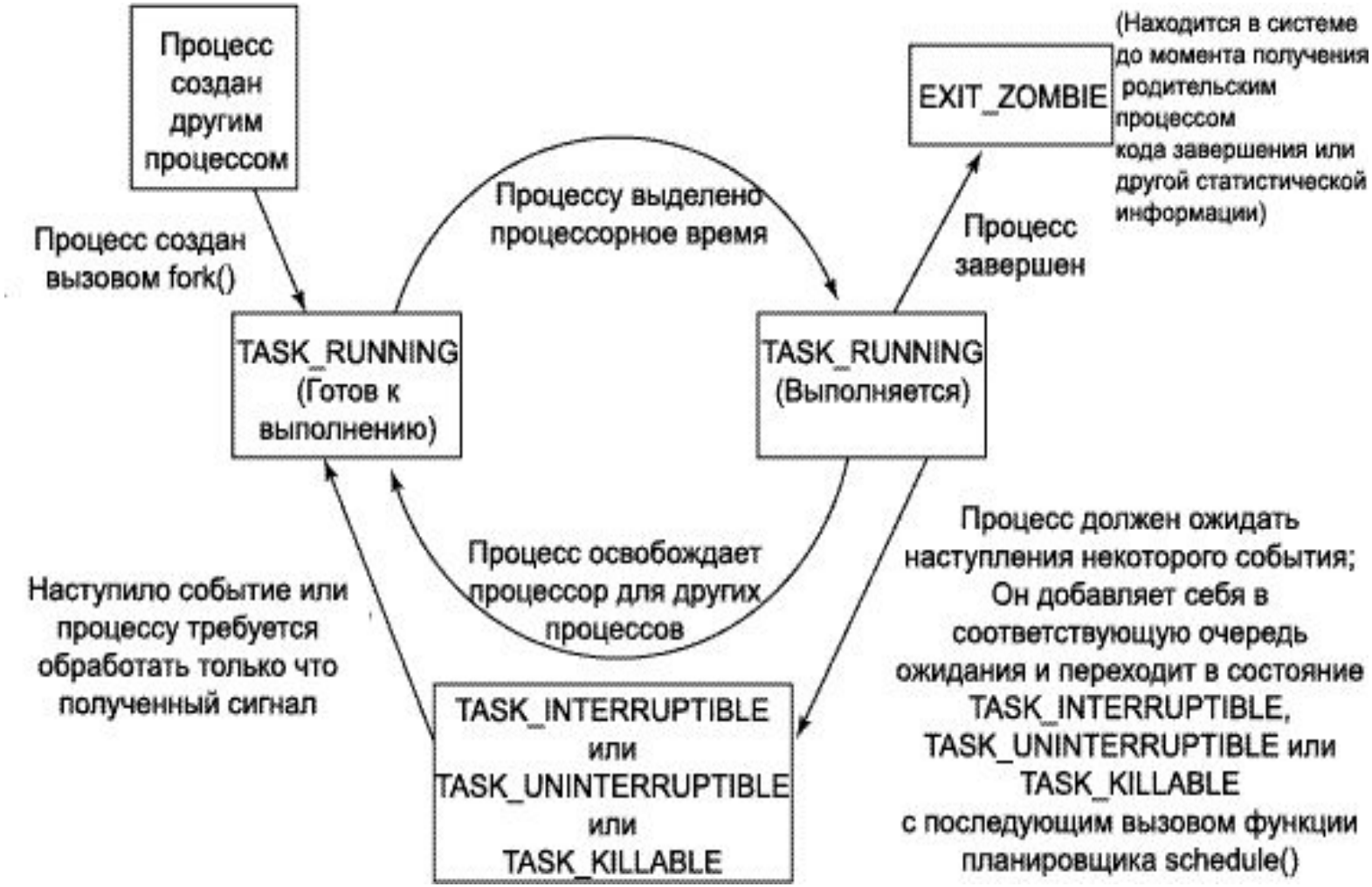
gid_t getgid(void); реальный групповой идентификатор

gid_t getegid(void); эффективный групповой идентификатор

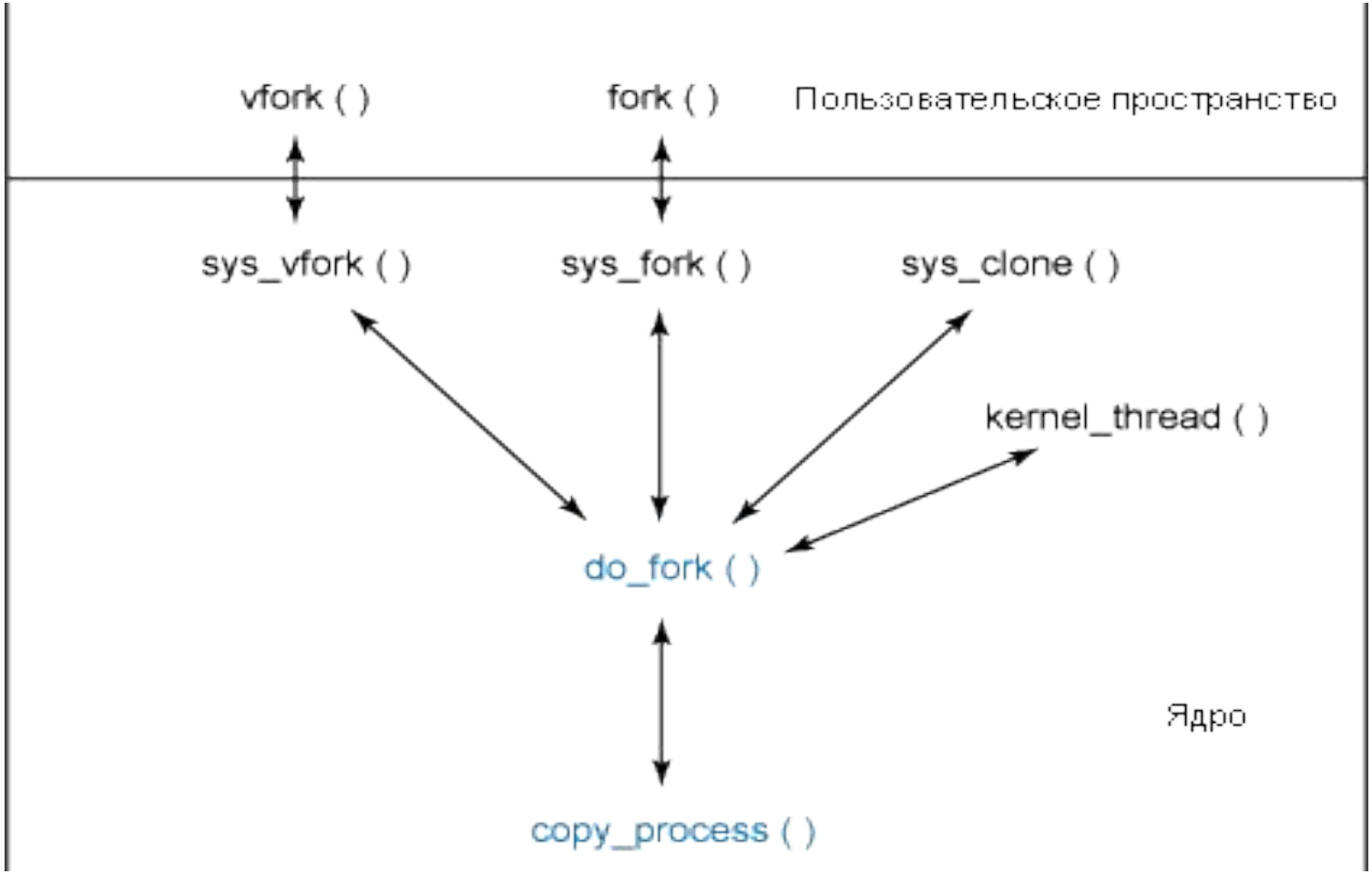
Состояния процесса

#define TASK_RUNNING 0 выполняется или готов к выполнению
#define TASK_INTERRUPTIBLE 1 приостановлен (ожидает события, может активизироваться сигналом)
#define TASK_UNINTERRUPTIBLE 2 приостановлен (ожидает события, не реагирует на сигналы)
#define __TASK_STOPPED 4 приостановлен (переходит по сигналу SIGSTOP, SIGTSTP ожидает сигнала SIGCONT)
#define __TASK_TRACED 8 трассировка процесса отладчиком
/* in tsk->exit_state */
#define EXIT_ZOMBIE 16 готов к уничтожению (дескриптор доступен)
#define EXIT_DEAD 32 уничтожен предком (дескриптор недоступен)
#define TASK_KILLABLE приостановлен (ожидает события, может активизироваться фатальным сигналом)

Диаграмма состояний



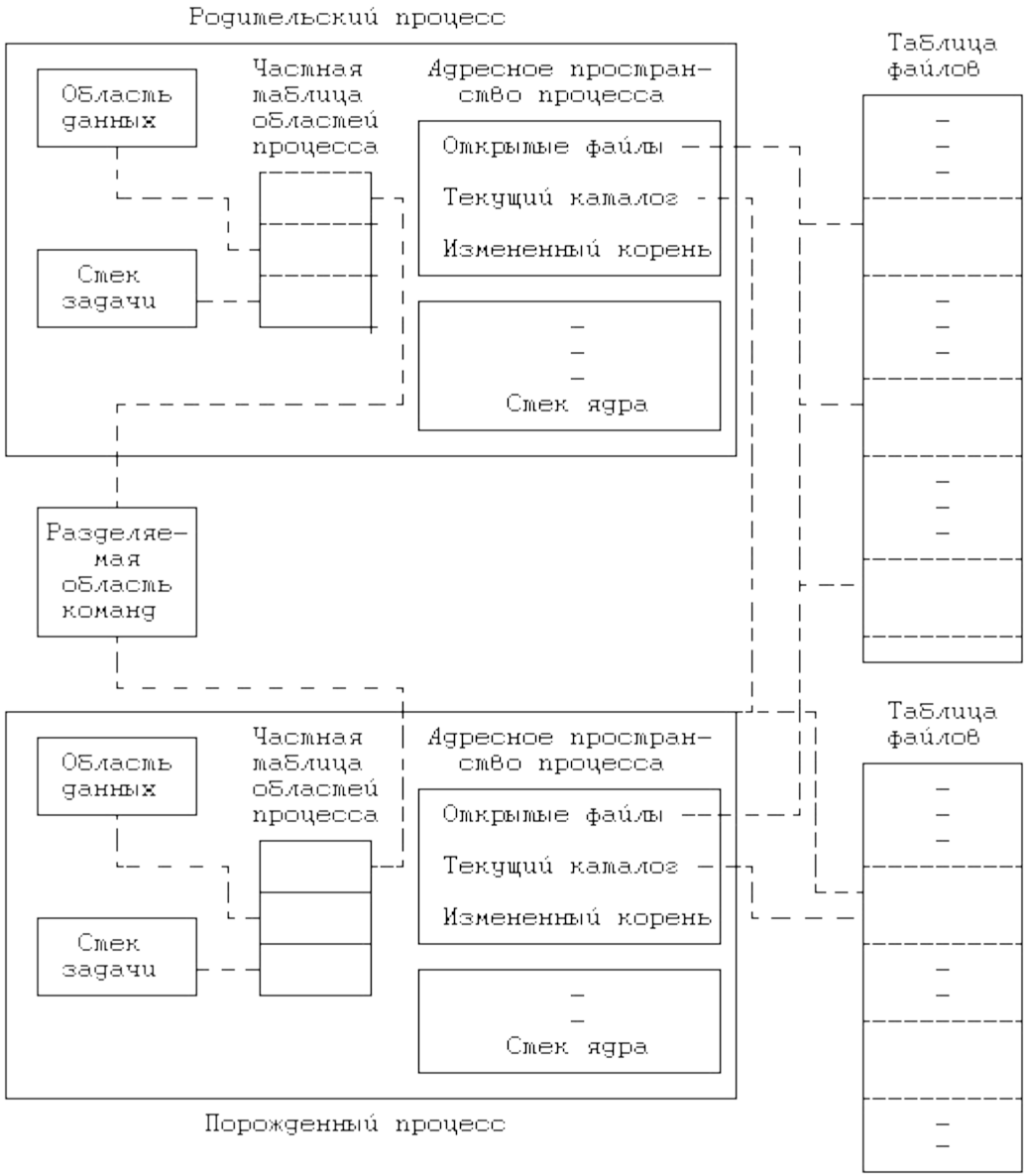
Создание процесса



Контексты процесса

- пользовательский контекст (области команд и данных, стек задачи, динамически выделяемая область, разделяемая память);
- системный контекст (дескриптор процесса, таблица страниц, стек ядра, таблица файлов)
- регистровый контекст (счетчик команд, регистр состояния процессора, указатель вершины стека ядра или задачи, регистры общего назначения)

Представление контекстов предка и потомка



Потомок наследует атрибуты предка

- Сегменты кода, данных и стека.
- Таблицу файлов.
- Рабочий и корневой каталоги.
- Реальный и эффективный идентификаторы пользователя и группы.
- Статический приоритет.
- Терминальное устройство.
- Маску сигналов.
- Разделяемые сегменты.

Потомок не наследует атрибуты предка

- Израсходованное время ЦП (оно обнуляется).
- Необработанные сигналы предка (они сбрасываются).
- Блокировки файлов (они снимаются).

Шаблон использования функции fork для создания процесса

pid=fork();

pid= идентификатор потомка для предка
0 для потомка
-1 ошибка

```
int pid;
if ((pid=fork( ) )<0)
  { /* ошибка -процесс не создан  */
   }
else if (pid==0)
  { /* операторы процесса потомка */
   }
else
 { /* операторы процесса предка */  }
```

Функции замены программы процесса

int execl (char* name, char* arg0, char* arg1, . . . , char* argn, (char*)0);

число параметров фиксировано

int execv (char* name, char* argv[ ]);

число параметров не известно

name - указатель на строку, содержащую описание пути расположения файла программы на диске. Если указывается только имя файла, то файл выбирается из текущего каталога.

arg0, . . . ,argn – параметры программы (функции main)

arg0 - имя вызываемого файла

Использование функции vfork() для создания процесса

vfork() отличается от fork тем, что родительский процесс блокируется до тех пор, пока дочерний процесс не вызовет **exec** или **_exit** (завершает процесс без об-
навления буферов потока и не закрывает открытые файлы).

Дочерний процесс разделяет всю память с родительским, включая стек, до тех пор, пока не вызовет *exec*

Разделение памяти при fork и vfork

```
// int common_variable;
static int create_process(void) {
pid_t pid; int status;
int common_variable = 0;
pid = fork();
//pid = vfork();
if (-1 == pid) { return errno; }
if (pid == 0) { common_variable = 1; exit(EXIT_SUCCESS); } //потомок
// предок
waitpid(pid, &status, 0); // ожидание завершения потомка
if (common_variable) { puts("vfork(): common variable изменилась."); }
else { puts("fork(): common variable не изменилась."); }
return EXIT_SUCCESS; }
int main(void) {
return create_process(); }
```

```
fork(): common variable не изменилась.
vfork(): common variable изменилась.
```

Шаблон замены программы потомка

```
pid_t pid;
if ((pid=vfork( ) )<0)
   { /* ошибка -процесс не создан  */
    }
else if (pid==0)
   { /* вызов программы для процесса потомка */
     execl(“prog”,”prog”);
    }
else
  { /* операторы процесса предка */ }
```

Задержка процесса

Активное ожидание:

void ndelay(unsigned long nanoseconds);

void udelay(unsigned long microseconds);

void mdelay(unsigned long milliseconds);

Пассивное ожидание:

```
#include <unistd.h>
int usleep(useconds_t usec);  мсек
int sleep(unsigned sec);          сек
#include <time.h>
int nanosleep(const struct timespec *req, struct
   timespec *rem);       мксек
struct timespec { time_t tv_sec; /* секунды */
long tv_nsec; /* наносекунды */ };
```

Завершение процесса

Когда процесс завершается производятся следующие действия:

- Все описатели открытого файла в процессе будут закрыты.
- 8 битов младшего разряда возвращающегося кода состояния сохранены в дескрипторе потомка, для передачи родительскому процессу;
- Любым дочерним процессам завершаемого процесса будет назначен новый родительский процесс.
- Сигнал SIGCHLD послан родительскому процессу.
- Если, процесс является лидером сеанса, который контролировал терминал управления, то сигнал SIGHUP будет послан каждому процессу сеанса, и терминал управления - будет отсоединен от этого сеанса.
- Если окончание процесса останавливает любой элемент группы этого процесса, то сигнал SIGHUP и сигнал SIGCONT будет послан каждому процессу в группе.
- Освобождается память, выделенная процессу.
- Процесс устанавливается в состояние TASK_ZOMBIE.

Функции завершения процесса

void exit(int exitCode); // завершение с указанием кода и вызова функции очистки

void _exit(int exitCode); // завершение с указанием кода

int atexit (void (*function) (void)) // регистрация функции очистки

```
void bye (void) {
puts (“До свидания");
}
int main (void) {
atexit (bye);
…
exit (EXIT_SUCCESS); // успешное завершение с вызовом функции bye
}
```

EXIT_FAILURE – неудачное завершение

Уничтожение процессов

int kill(pid_t pid, int signum);

pid > 0 Сигнал отправляется процессу с идентификатором pid.

pid < -1 Сигнал посылается всем процессам, принадлежащим группе с pgid, равным -pid.

pid = 0 Сигнал отправляется всем процессам группы, к которой относится текущий процесс.

pid = -1 Сигнал посылается всем процессам системы за исключением инициализирующего процесса (init). Это применяется для полного завершения системы.

signum:

SIGTERM – «вежливое » уничтожение (можно перехватить или заблокировать)

SIGKILL – безусловное уничтожение (нельзя перехватить или заблокировать)

Задание к лабораторной работе 3

Разработайте программу, которая порождает 2 потомка. Первый потомок порождается с помощью **fork**, второй с помощью **vfork** с последующей заменой на другую программу. Все 3 процесса должны вывести в один файл свои атрибуты с предварительным указанием имени процесса (например: Предок, Потомок1, Потомок2). Имя выходного файла задается при запуске программы. Порядок вывода атрибутов в файл должен определяться задержками процессов, которые задаются в качестве параметров программы.

Файл с атрибутами процесса

Задержки: 1 5 10
Предок:
идентификатор процесса=6555
идентификатор предка=3108
идентификатор сессии процесса=3108
идентификатор группы процессов=6555
реальный идентификатор пользователя=1000
эффективный идентификатор пользователя=1000
реальный групповой идентификатор=1000
эффективный групповой идентификатор=1000
Потомок1:
идентификатор процесса=6556
идентификатор предка=1470
идентификатор сессии процесса=3108
идентификатор группы процессов=6555
реальный идентификатор пользователя=1000
эффективный идентификатор пользователя=1000
реальный групповой идентификатор=1000
эффективный групповой идентификатор=1000
Потомок2:
идентификатор процесса=6557
идентификатор предка=1470
идентификатор сессии процесса=3108
идентификатор группы процессов=6555
реальный идентификатор пользователя=1000
эффективный идентификатор пользователя=1000
реальный групповой идентификатор=1000
эффективный групповой идентификатор=1000