ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лекция № 02 — Процессы

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2024

2024.02.22

Оглавление

Что такое ЭВМ (компьютер)?

Компьютер — это машина для обработки битов.

Бит — это отдельная единица компьютерного хранилища информации, которая может принимать два альтернативных значения — мы их обычно зовем 0 и 1.

Компьютеры используются для обработки информации, но вся информация при этом представляется в виде битов. В этом случае бит — наименьшая единица информации.

Интерпретация бит и их последовательностей может быть различной.

Наборы битов могут представлять символы, цифры или любую другую информацию.

Люди интерпретируют эти биты как информацию, в то время как компьютеры просто манипулируют битами, которые представляют собой ***отображение состояний линий и выводов микросхем*** (уровни) на значения 0 и 1.

Состояние линий и выводов микросхем

H — High (высокий уровень)

L — Low (низкий уровень)

Байты

Блок из N бит, которые являются ***наименьшим адресуемым количеством информации*** в памяти компьютера, называется «***байтом***».

Современные компьютеры получают доступ к памяти в виде 8-битных блоков.

Таким образом байтом обычно называется 8-битное количество бит.

Основная память компьютера — это массив байтов, каждый из которых имеет отдельный адрес памяти.

Первый адрес байта равен 0, а последний адрес зависит от используемого аппаратного и программного обеспечения.

Байты организуются в группы.

Интерпретация данных и типизация

Байт можно интерпретировать как двоичное число. Двоичное число 01010101 равно десятичному числу 85 (64 + 16 + 4 + 1).

Число 85 может быть частью большего числа в компьютере.

Число 85 может интерпретироваться как машинная инструкция, в этом случае компьютер помещает значение регистра **`rbp`** в стек времени выполнения.

Число 85 также можно интерпретировать как символ — заглавную букву «U».

Буква «U» может быть частью символьной строки в памяти.

Биты — квант информации (1 или 0).
Тетрада — группа из 4-х бит (0...15).
Байты — квант адресуемой памяти (обычно 8 бит — 0...255). Байт можно интерпретировать как:
- машинная инструкция (55h — push rbp);
- как символ (55h— буква «U»), как часть мультибайтного символа или символьной строки;
- двоичное число или часть большего числа.

Слова — группа из 2^k байт (наследие 8086/88).
BigEndian — адресуется старший байт (коммуникации, не x86)
LittleEndian — адресуется младший байт (x86) (Не путать с MSB/LSB — Most/Least Significant Bit)

Hexadecimal	Decimal	Binary	Adress	BE	LE
0x6B	107	0101 0101	70ab 5200	6 B	6 B
0xAF	175	1010 1111	70ab 5201	A F	A F
0xA7FE	43 006	1010 0111 1111 1110	70ab 5202	A 7	F E
			70ab 5203	F E	A 7
0x2C03 A7FE	738 437 118	0010 1100 0000 0011 1010 0111 1111 1110	70ab 5204	2 C	F E
			70ab 5205	0 3	A 7
			70ab 5206	A 7	0 3
			70ab 5207	F E	2 C

MSB ← 10101111 → LSB

Кроме вышеперечисленных есть другие типы данных, которые могут быть представлены в двоичном виде.

В области телекоммуникаций и компьютерных технологий используется стандарт, описывающий структуры данных для представления, кодирования, передачи и декодирования данных — **ASN.1** (Abstract Syntax Notation One).

Начиная с 1995 года, существенно пересмотренный ASN.1 описывается стандартом X.680.

В России ASN.1 стандартизирован по:

ГОСТ Р ИСО/МЭК 8824-1-2001

ГОСТ Р ИСО/МЭК 8825-93

Аппаратные возможности компьютеров

Один, отдельно взятый, процессор (процессорное ядро), в один момент времени, может исполнять только одну программу (XX-DOS).

Но к компьютерам предъявляются более широкие требования. В связи с этим современные процессоры имеют ***мультизадачные возможности*** — процессор выполняет какую-то одну программу (процесс или задач). По истечении некоторого времени (микросекунды), ОС переключает процессор на другую программу. При этом все регистры текущей программы (состояние, контекст) сохраняются. Через некоторое время вновь передается управление этой программе. Программа при этом не замечает каких либо изменений — для нее процесс переключения незаметен.

Для того чтобы программа не могла каким либо образом нарушить работоспособность системы или других программ, предусмотрены механизмы защиты на основе *уровней привилегий*.

Уровень привилегий — это степень использования ресурсов процессора. Всего таких уровней четыре и они имеют номера от 0 до 3.

Уровень 0 - самый привилегированный. На этом уровне, программе «можно всё».

Уровень 1 - менее привилегированный – запреты, установленные на уровне 0, действуют для уровня 1.

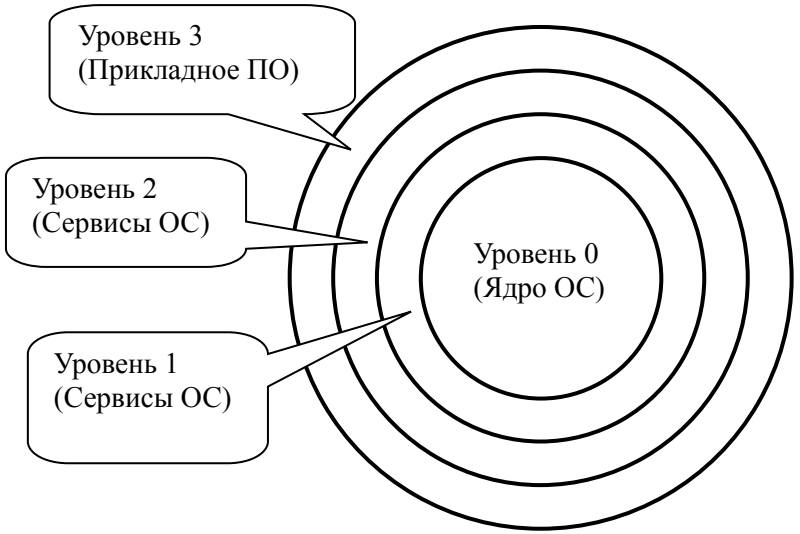
Уровень 2 - ещё менее привилегированный.

Уровень 3 - имеет самый низкий приоритет.

Классическая схема работы программ по уровням привилегий

Классическая схема работы программ по уровням привилегий имеет вид:

уровень 0: ядро операционной системы;
уровень 1: драйверы ОС;
уровень 2: интерфейс ОС;
уровень 3: прикладные программы.



Использование всех четырех уровней привилегий не является необходимым. Существующие системы, спроектированные с меньшим количеством уровней, могут просто игнорировать другие допустимые уровни.

***NIXы** и **Windows**, например, используют только два уровня привилегий:
0 – ядро системы;
3 – все остальное.
IBM OS/2 использует уровни:
0 – ядро системы;
2 – процедуры ввода-вывода;
3 – прикладные программы.

Процессы

Многозадачность — это один из основных параметров всех современных ВС.

Фундаментальным понятием современных ВС является понятие ***процесса*** — основного динамического объекта, над которыми ОС выполняет определенные действия.

По ходу работы программы ВС обрабатывает различные команды и преобразует значения переменных. Для этого ОС должна выделить определенное количество оперативной памяти, закрепить за ней определенные устройства ВВ или файлы (откуда должны поступать входные данные и куда нужно доставить полученные результаты), то есть зарезервировать определенные ресурсы из общего числа ресурсов всей ВС.

Количество и конфигурация ресурсов с течением времени обычно изменяются.

Для описания таких активных объектов внутри вычислительной системы используется термин **«процесс»** — программа, загруженная в память и выполняющаяся.

Компьютерная программа — это всего лишь пассивная совокупность инструкций.

Процесс — непосредственное выполнение программных инструкций.

Иногда процессом называют выполняющуюся программу и все её элементы:

- адресное пространство;
- глобальные переменные;
- регистры;
- стек;
- открытые файлы и прочие ресурсы.

Понятие процесса охватывает:

1) некоторую совокупность исполняющихся команд;

2) совокупность ассоциированных с процессом ресурсов — выделенная для исполнения память или адресное пространство, стеки, используемые файлы, устройства ввода-вывода, и т. д (прикладной контекст, системный контекст).

3) текущий момент выполнения — значения регистров и программного счетчика, состояние стека, значения переменных (регистровый контекст).

Взаимно однозначного соответствия между процессами и программами нет.

Обычно в рамках программы можно организовывать более одного процесса и наоборот.

Процесс находится под управлением операционной системы и поэтому в нем может выполняться часть кода ее ядра, которая не находится в исполняемом файле. Это происходит как в случаях, специально запланированных авторами программы, например, при использовании системных вызовов и при обработке внешних прерываний.

Процесс — это находящаяся под управлением ОС совокупность некоторого набора исполняющихся инструкций, ассоциированных с процессом ресурсов ВС и текущего момента выполнения.

Все, что выполняется в вычислительных системах (программы пользователей и определенные части операционных систем), организовано в виде набора процессов.

Состояния процесса

На однопроцессорной (одноядерной) компьютерной системе в каждый момент времени может исполняться только один процесс.

Для мультипрограммных вычислительных систем ***псевдопараллельная*** обработка нескольких процессов достигается с помощью **переключения процессора с одного процесса на другой** — пока один процесс выполняется, остальные ждут своей очереди на доступ к процессору.

Каждый процесс может находиться, как минимум, в двух состояниях:

- процесс исполняется;

- процесс не исполняется.

Процесс, находящийся в состоянии «**процесс исполняется**», может через некоторое время завершиться или быть приостановлен операционной системой и переведен в состояние «**процесс не исполняется**». Приостановка происходит по одной из двух причин:

1) для его дальнейшей работы потребовалось возникновение какого-либо события (например, завершения операции ввода-вывода);

2) истек временной интервал, отведенный операционной системой для работы этого процесса.

После этого операционная система по определенному алгоритму выбирает для исполнения один из процессов, находящихся в состоянии «**процесс не исполняется**», и переводит его в состояние «**процесс исполняется**».

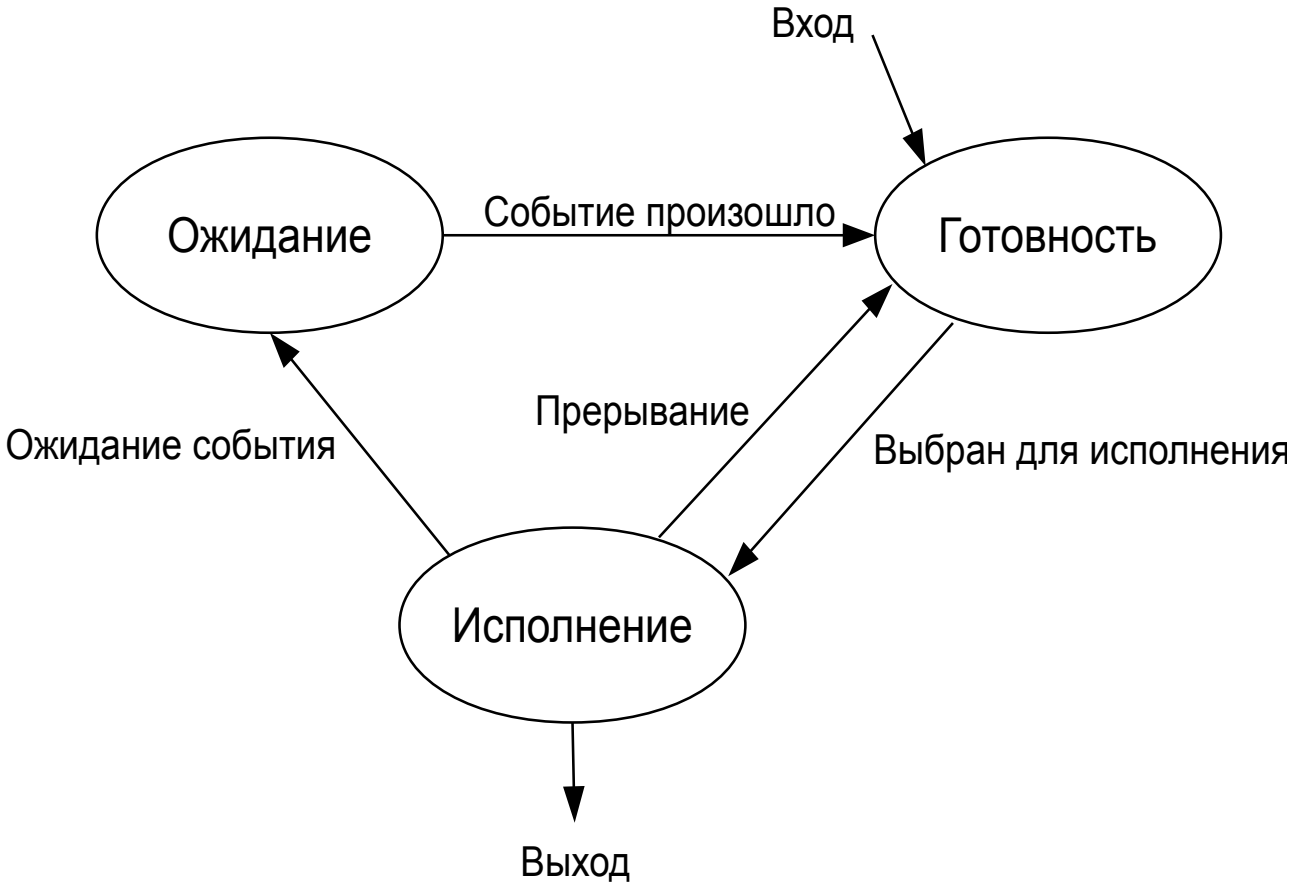
Процесс, выбранный для исполнения, может все еще ждать события, из-за которого он был приостановлен, и реально к выполнению не готов.

Чтобы это учесть, состояние «**процесс не исполняется**» разбивается на два новых:

- готовность;

- ожидание.

Всякий новый процесс, появляющийся в системе, попадает в состояние **«готовность»**. ОС, пользуясь каким-либо алгоритмом планирования, выбирает один из готовых процессов и переводит его в состояние **«исполнение»**.

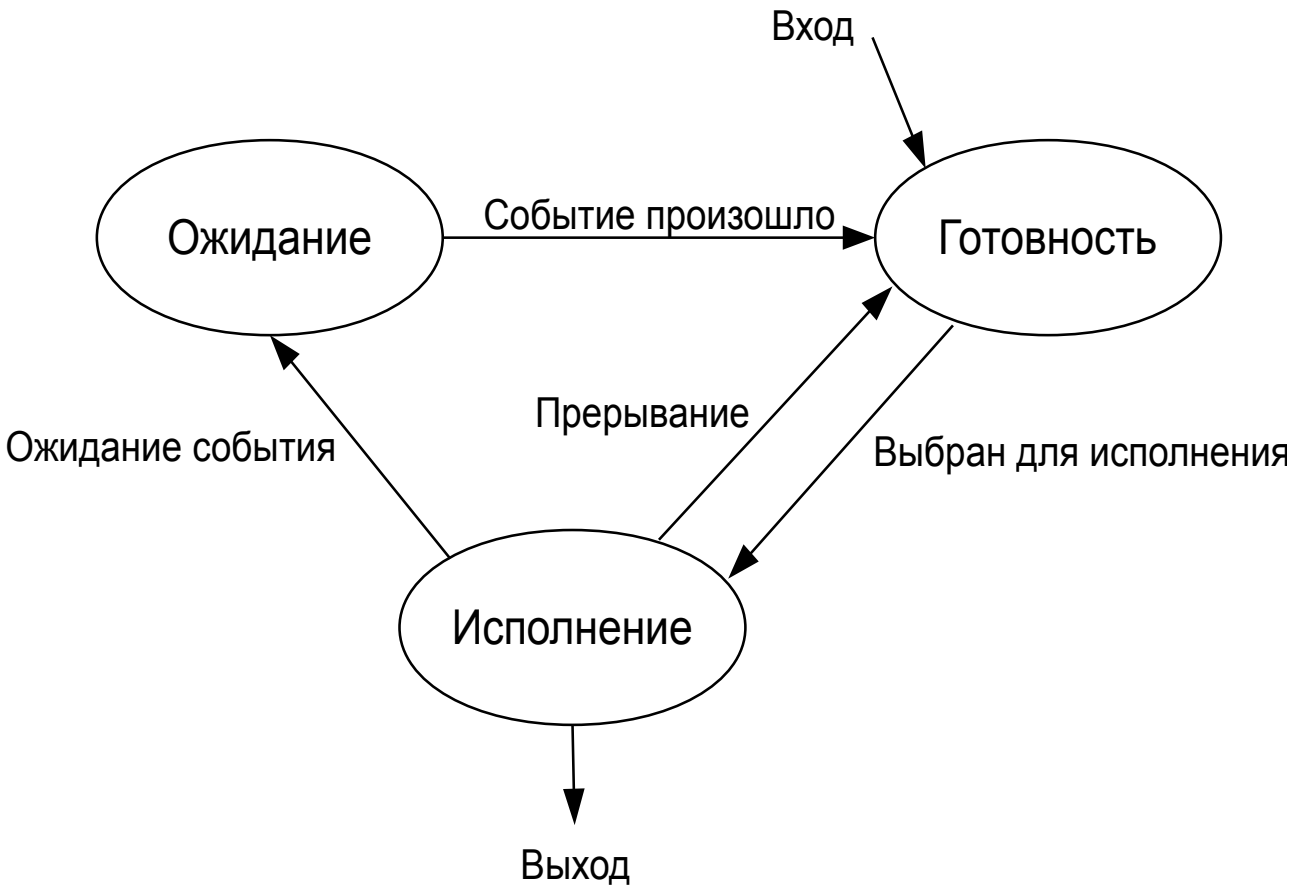


В состоянии **«исполнение»** происходит непосредственное выполнение программного кода процесса. Покинуть это состояние процесс может по трем причинам:

1) процесс заканчивает свою деятельность;

2) процесс не может продолжать свою работу, пока не произойдет некоторое событие, и операционная система переводит его в состояние **«ожидание»**;

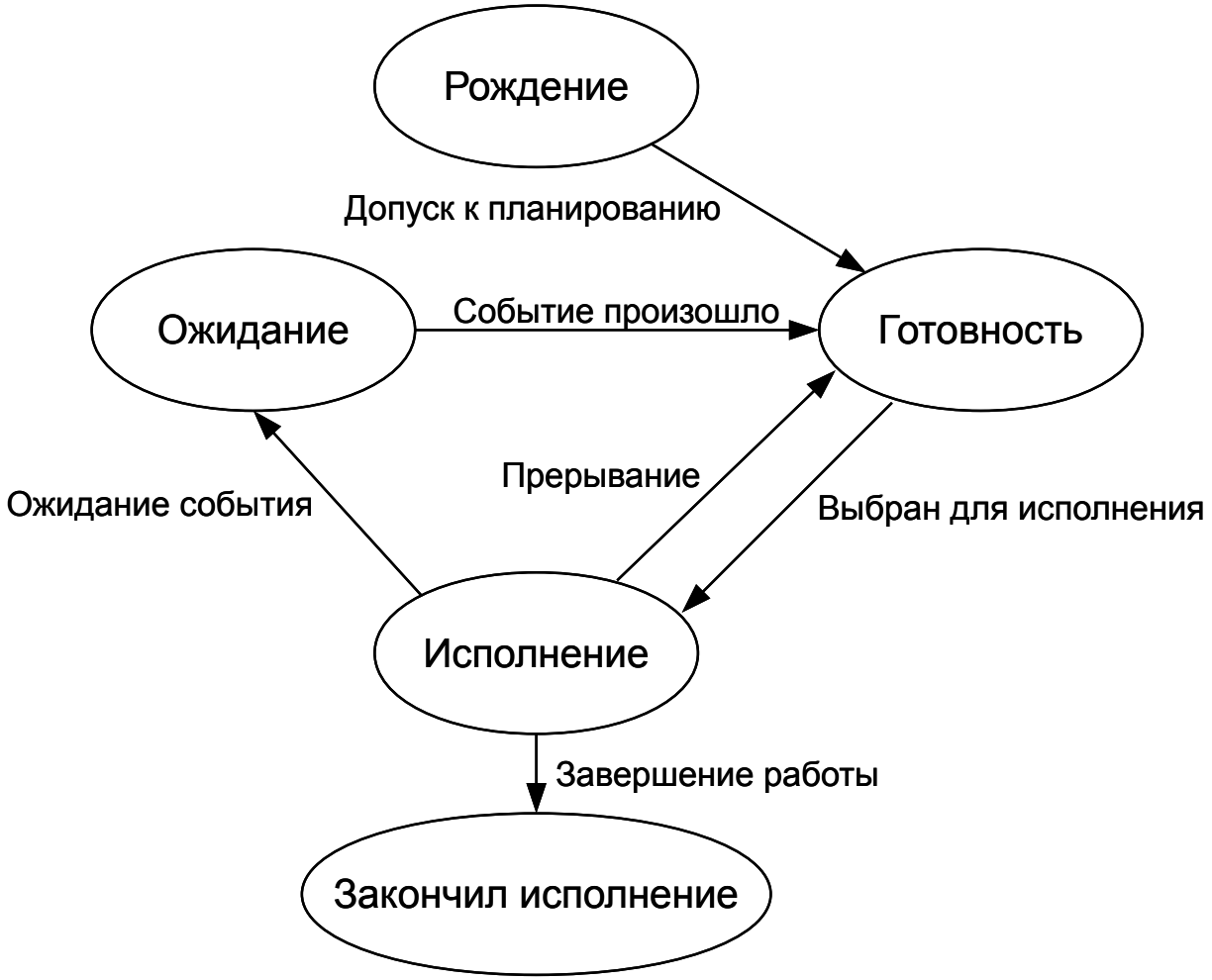
3) процесс возвращается в состояние **«готовность»** в результате возникновения прерывания в вычислительной системе (например, прерывания от таймера по истечении дозволенного времени выполнения).



В реальной ОС вводится еще два состояния:

- рождение;

- закончил исполнение.



Для появления в вычислительной системе процесс должен пройти через состояние **«рождение»**.

При рождении процессу выделяются адресное пространство, в которое загружается программный код процесса, стек и системные ресурсы, устанавливается начальное значение программного счетчика этого процесса и т. д.

После этого родившийся процесс переводится в состояние «готовность».

При завершении своей деятельности процесс из состояния исполнение попадает в состояние **«закончил исполнение»**.

Операции над процессами

Процесс не может сам перейти из одного состояния в другое.

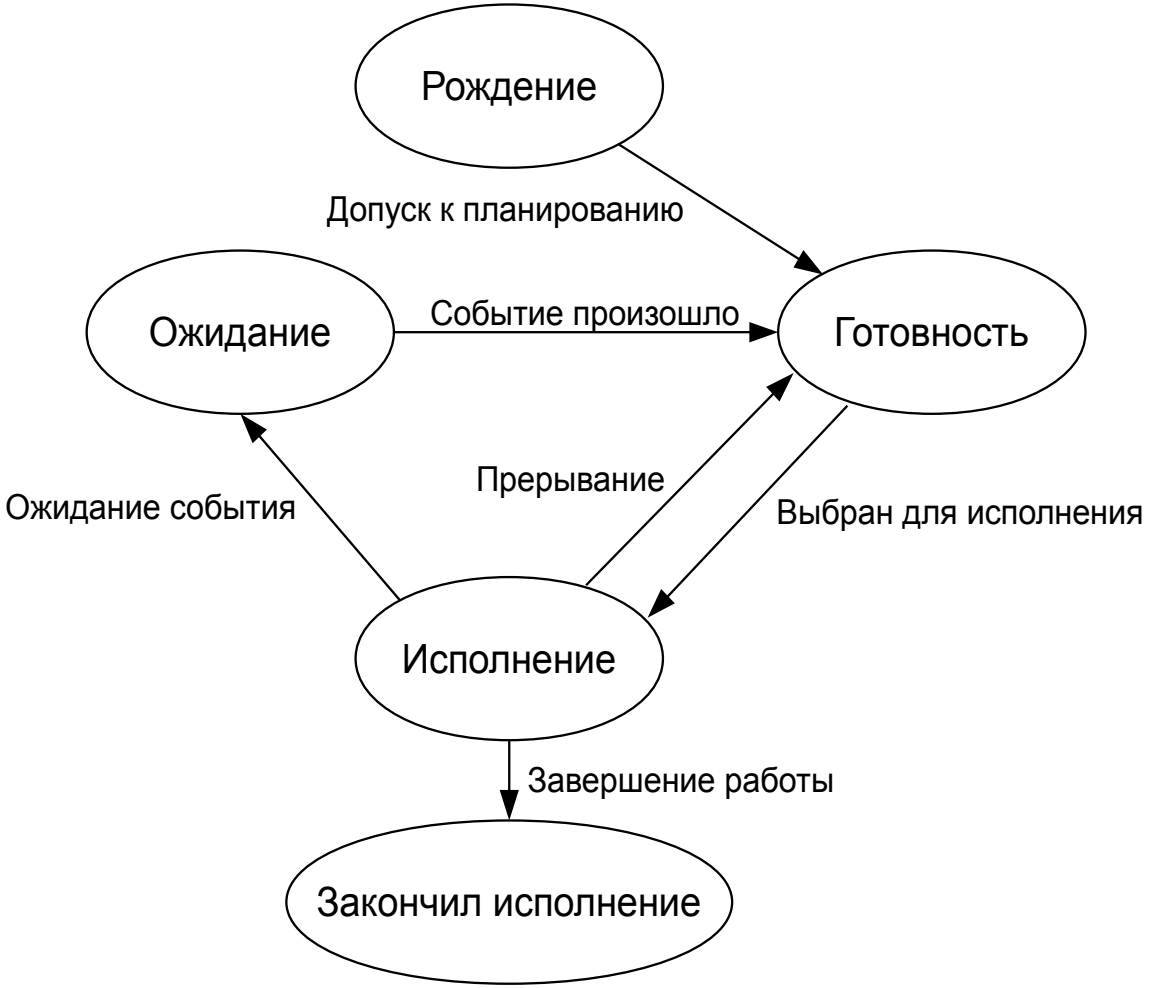
Изменением состояния процессов занимается ОС, совершая над ними операции:

Создание — завершение

Приостановка — запуск

Блокирование — разблокирование

Изменение приоритета



Операции создания и завершения процесса являются одноразовыми, так как применяются к процессу не более одного раза (некоторые системные процессы никогда не завершаются при работе вычислительной системы).

Все остальные операции, связанные с изменением состояния процессов, будь то запуск или блокировка, как правило, являются многоразовыми.

Process Control Block – блок управления процессом и контекст процесса

Для того чтобы операционная система могла выполнять операции над процессами, каждый процесс представляется в ней некоторой структурой данных.

Эта структура содержит информацию, специфическую для данного процесса:

- состояние, в котором находится процесс;
- программный счетчик (адрес команды, которая будет выполнена следующей);
- содержимое регистров процессора;
- данные, необходимые для планирования использования процессора и управления памятью (приоритет , размер и расположение адресного пространства и т. д.);
- учетные данные (идентификационный номер процесса, какой пользователь инициировал его работу, общее время использования процессора данным процессом и т. д.);
- информацию об устройствах ввода-вывода, связанных с процессом (например, какие устройства закреплены за процессом, таблицу открытых файлов).

Во многих ОС информация, характеризующая процесс, хранится не в одной, а в нескольких связанных структурах данных. Эти структуры могут иметь различные наименования, содержать дополнительную информацию или, наоборот, лишь часть описанной информации (аппаратный вариант — TSS[1]).

---

1 Task State Segment — https://pdos.csail.mit.edu/6.828/2005/readings/i386/s07_01.htm

Блок управления процессом содержит всю информацию, необходимую для выполнения операций над процессом — **он является моделью процесса для ОС**.

Любая операция, производимая ОС над процессом, вызывает определенные изменения в БУП.

Информацию, для хранения которой предназначен блок управления процессом, можно разделить на две части:

- регистровый контекст — содержимое всех регистров процессора (включая значение программного счетчика);

- системный контекст — все остальное.

Знания регистрового и системного контекстов процесса достаточно для того, чтобы управлять его поведением в операционной системе, совершая над ним операции, однако недостаточно, чтобы полностью характеризовать процесс.

Пользовательский контекст

Операционную систему не интересует, чем именно занимается процесс, т. е. какой код и какие данные находятся в его адресном пространстве.

С точки зрения пользователя, наоборот, наибольший интерес представляет содержимое адресного пространства процесса, возможно наряду с регистровым контекстом, определяющее последовательность преобразования данных и полученные результаты.

Пользовательский контекст — код и данные, находящиеся в адресном пространстве процесса.

Контекст процесса — совокупность регистрового, системного и пользовательского контекстов.

В любой момент времени процесс полностью характеризуется своим контекстом.

В контексте архитектуры x86 (IA32) используется термин задача (**task**), который в некоторой степени, является эквивалентом понятия процесс.

Задача — это некоторая самостоятельная последовательность команд (процесс), которая выполняется в своём окружении.

Основные параметры, которыми характеризуется окружение задачи:

- состояние регистров общего назначения (E[ABCD]X, E[SD]I);
- состояние селекторных (сегментных) регистров (DS, CS, SS,ES, FS, GS);
- адресное пространство, которое характеризуется регистром CR3;
- состояние регистров математического сопроцессора и расширений (MMX, SSE*, XMM, AVX*, ...);
- карта портов ввода-вывода.

Поскольку в большинстве случаев количество задач (процессов) намного больше количества процессоров (ядер), многозадачность реализуется путём быстрого **переключения задач**. Именно благодаря этому создаётся ощущение, что все задачи работают одновременно.

Устройство ядра ОС

1) «собственно ядро»;
2) драйверы устройств;
3) системные вызовы

«Собственно ядро» — функции управления памятью и процессами. Переключение процессов — это важнейший момент нормального функционирования системы.

Драйвера — это специальные программы, обеспечивающие работу устройств компьютера. В некоторых OS (FreeBSD) предусматриваются механизмы прерывания работы драйверов по истечении какого-то времени. Тем не менее, можно написать драйвер под FreeBSD или Linux, который полностью заблокирует работу системы — это следствие двухуровневой защиты ==> драйвера надо тщательно программировать.

Общая производительность системы, в основном, зависит именно от драйверов.

Системные вызовы — это интерфейс между процессами и ядром. Никаких других методов взаимодействия процессов с устройствами компьютера быть не должно.

Системных вызовов достаточно много, в Linux 6.6 их около 450[2]/370[3], во FreeBSD их порядка 500, причем большей частью они совпадают, соответствуя стандарту POSIX (стандарт, описывающий системные вызовы в UNIX). Разница заключается в передаче параметров (ABI).

---

2 /usr/include/asm/unistd_32.h
3 /usr/include/asm/unistd_64.h

Прикладным программам абсолютно безразлично, как системные вызовы реализуются в ядре. Это облегчает обеспечение совместимости с существующими системами.

Основные системные вызовы:

- системные вызовы для работы с каталогами;
- системные вызовы для работы с файлами (ввод/вывод);
- системные вызовы для работы с памятью.

В прикладных программах обработки данных наиболее часто используются системные вызовы данных типов. В linux системные вызовы объявляются в заголовке:

```
$ cat /usr/include/asm/unistd.h                                  # с-комментарии удалены
#ifndef _ASM_X86_UNISTD_H
#define _ASM_X86_UNISTD_H
...
#define __X32_SYSCALL_BIT  0x40000000

# ifdef __i386__
#  include <asm/unistd_32.h>
# elif defined(__ILP32__)
#  include <asm/unistd_x32.h>
# else
#  include <asm/unistd_64.h>
# endif

#endif /* _ASM_X86_UNISTD_H */
```

```
$ cat /usr/include/asm/unistd_32.h | grep '__NR_' | wc
    438    1314   11793
$ cat /usr/include/asm/unistd_32.h
#ifndef _ASM_X86_UNISTD_32_H
#define _ASM_X86_UNISTD_32_H 1
#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
#define __NR_link 9
#define __NR_unlink 10
#define __NR_execve 11
#define __NR_chdir 12
#define __NR_time 13
#define __NR_mknod 14
#define __NR_chmod 15
#define __NR_lchown 16
#define __NR_break 17
#define __NR_oldstat 18
#define __NR_lseek 19
#define __NR_getpid 20
...
#endif /* _ASM_UNISTD_32_H */
```

```
$ cat /usr/include/unistd_64.h | grep '__NR_' | wc
    360    1080    9604
$ cat /usr/include/unistd_64.h
#ifndef _ASM_X86_UNISTD_64_H
#define _ASM_X86_UNISTD_64_H 1
#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_stat 4
#define __NR_fstat 5
#define __NR_lstat 6
#define __NR_poll 7
#define __NR_lseek 8
#define __NR_mmap 9
#define __NR_mprotect 10
#define __NR_munmap 11
#define __NR_brk 12
#define __NR_rt_sigaction 13
#define __NR_rt_sigprocmask 14
#define __NR_rt_sigreturn 15
#define __NR_ioctl 16
define __NR_pread64 17
#define __NR_pwrite64 18
#define __NR_readv 19
#define __NR_writev 20
...
#endif /* _ASM_UNISTD_64_H */
```

Системные вызовы управления процессами

`fork(2)` — создает дочерний процесс
`execve(2)` — выполнить программу
`exit()` — функция, завершающая работу программы
`wait(2)` — ожидает завершения процесса
`waitpid(2)` — ожидает завершения процесса
`waitid(2)` — ожидает завершения процесса
`environ(7)` – пользовательское окружение

fork — создает дочерний процесс

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

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.

Под Linux **fork** реализован с помощью "копирования страниц при записи" (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 неудаче родительскому процессу возвращается **-1**, процесс-потомок не создается, а значение **errno** устанавливается должным образом.

Ошибки

EAGAIN — **fork** не может выделить достаточно памяти для копирования таблиц страниц родителя и для выделения структуры описания процесса-потомка.

ENOMEM — **fork** не может выделить необходимые ресурсы ядра, потому что памяти слишком мало.

Пример

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    pid_t pid, ppid, chpid;
    int a = 0;
    chpid = fork();
// При успешном создании нового процесса с этого места псевдопараллельно
// начинают работать 2 процесса: старый и новый
// Перед выполнением следующего выражения a в обоих процессах равно 0
    a = a + 1;
// Узнаем идентификаторы текущего и родительского процесса в каждом из них
    pid  = getpid();
    ppid = getppid();
// Печатаем значения PID, PPID и вычисленное значение a в каждом из процессов
    printf("My pid = %d, my ppid = %d, result = %d\n", (int)pid, (int)ppid, a);
    return 0;
}
```

Вывод

My pid = **128969**, my ppid = 116473, result = 1

My pid = 128970, my ppid = **128969**, result = 1

Пример

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/wait.h>

_Noreturn int child_foo(void);
int child_status;

int main() {

    fprintf(stdout, "Родительский процесс стартовал ...\n");
    pid_t pid = fork();
    if (pid == -1) {
        fprintf(stdout, "Ошибка, код ошибки - %d\n", errno);
    }
    if (pid == 0) { // дочерний процесс
        fprintf(stdout, "Это дочерний процесс. Вызываем child_foo()...\n");
        child_foo();
    }
    fprintf(stdout, "Родительский процесс продолжает выполнение\n");
    wait(&child_status);
    fprintf(stdout, "Дочерний процесс завершился с кодом завершения %d\n",
            WEXITSTATUS(child_status));
    exit(0);
}
```

```
_Noreturn int child_foo(void) {

    fprintf(stdout, "%s()\n", __func__);
    exit(123);
}
```

Выход

```
Родительский процесс стартовал ...
Родительский процесс продолжает выполнение
Это дочерний процесс. Вызываем child_foof()...
child_foo()
Дочерний процесс завершился с кодом завершения 123
```

execve — выполнить программу

```
#include <unistd.h>

int execve(const char *filename,
           char *const argv[],
           char *const envp[]);
```

execve( ) выполняет программу, заданную параметром **filename**.

Программа должна быть либо двоичным исполняемым файлом, либо скриптом, начинающимся со строки вида

```
"#! интерпретатор [аргументы]"
```

В последнем случае интерпретатор — это правильный путь к исполняемому файлу, который не является скриптом — этот файл будет выполнен как **интерпретатор [arg] filename**.

argv — это массив строк, аргументов новой программы.

envp — это массив строк в формате **key=value**, которые передаются новой программе в качестве окружения (*environment*).

Как **argv**, так и **envp** завершаются нулевым указателем.

К массиву аргументов и к окружению можно обратиться из функции **main( )**, которая имеет прототип

```
int main(int, char *[], char *[]);
```

execve( ) не возвращает управление при успешном выполнении, а код, данные, bss и стек вызвавшего процесса перезаписываются кодом, данными и стеком загруженной программы.

Новая программа также наследует от вызвавшего процесса его идентификатор (**pid**) и открытые файловые дескрипторы, *на которых не было флага «close-on-exec»* (COE).

Сигналы, ожидающие обработки, удаляются.

Переопределённые обработчики сигналов возвращаются в значение по умолчанию.

Обработчик сигнала **SIGCHLD** (когда установлен в **SIG_IGN**) может быть сброшен или не сброшен в **SIG_DFL**.

Если текущая программа выполнялась под управлением **ptrace**, то после успешного **execve( )** ей посылается сигнал **SIGTRAP**.

Если на файле программы **filename** установлен `setuid`-бит, то фактический идентификатор пользователя вызывавшего процесса меняется на идентификатор владельца файла программы.

Точно так же, если на файле программы установлен `setgid`-бит, то фактический идентификатор группы устанавливается в группу файла программы.

Если исполняемый файл является динамически-скомпонованным файлом в формате **a.out**, содержащим заглушки для вызова совместно используемых библиотек, то в начале выполнения этого файла вызывается динамический компоновщик **ld.so**(8), который загружает библиотеки и компонует их с исполняемым файлом.

Если исполняемый файл является динамически-скомпонованным файлом в формате **ELF**, то для загрузки совместно используемых библиотек используется интерпретатор, указанный в сегменте **PT_INTERP**. Обычно это **/lib/ld-linux.so.1** для программ, скомпилированных под Linux **libc** версии 5, или же **/lib/ld-linux.so.2** для программ, скомпилированных под GNU **libc** версии 2.

Возвращаемое значение

При успешном завершении **execve( )** не возвращает управление.

При ошибке возвращается **-1**, а значение **errno** устанавливается должным образом.

Коды ошибок

EACCES — интерпретатор файла или скрипта не является обычным файлом.

EACCES — нет прав на выполнение файла, скрипта или ELF-интерпретатора.

EACCES — файловая система смонтирована с флагом **noexec**.

EPERM — файловая система смонтирована с флагом **nosuid**, пользователь не является суперпользователем, а на файле установлен бит **setuid** или **setgid**.

EPERM — процесс работает под отладчиком, пользователь не является суперпользователем, а на файле установлен бит **setuid** или **setgid**.

E2BIG — список аргументов слишком велик.

EFAULT — **filename** указывает за пределы доступного адресного пространства.

ENOEXEC — исполняемый файл в неизвестном формате, для другой архитектуры, или же встречены какие-то ошибки, препятствующие его выполнению.

ENAMETOOLONG — **filename** слишком длинное.

ENOENT — файл **filename**, или интерпретатор скрипта или ELF-файла не существует, или же не найдена разделяемая библиотека, требуемая файлу или интерпретатору.

ENOMEM — недостаточно памяти в ядре.

ENOTDIR — компонент пути **filename**, или интерпретатору скрипта или ELF-интерпретатору не является каталогом.

EACCES — нет прав на поиск в одном из каталогов по пути к **filename**, или имени интерпретатора скрипта или ELF-интерпретатора.

ELOOP — слишком много символьных ссылок встречено при поиске **filename**, или интерпретатора скрипта или ELF-интерпретатора.

ETXTBSY — исполняемый файл открыт для записи одним или более процессами.

EIO — произошла ошибка ввода-вывода.

ENFILE — достигнут системный лимит на общее количество открытых файлов.

EMFILE — процесс уже открыл максимально доступное количество открытых файлов.

EINVAL — исполняемый файл в формате ELF содержит более одного сегмента **PT_INTERP** (то есть, в нем указано более одного интерпретатора).

EISDIR — ELF-интерпретатор является каталогом.

ELIBBAD — ELF-интерпретатор имеет неизвестный формат.

Замечания

1) **SUID** и **SGID** процессы не могут быть оттрассированы **ptrace()**.
2) Linux игнорирует **SUID** и **SGID** биты на скриптах.

Первая строка (строка с **#!**) исполняемого скрипта не может быть длиннее 127 символов.

`environ` — среда пользователя

```
extern char **environ; // массив указателей
```

Переменная **`environ`** указывает на массив указателей на строки, называемый «окружением». Последний указатель в этом массиве имеет значение **`NULL`**.

Этот массив строк предоставляется процессу вызовом **`execve(2)`** при запуске новой программы по умолчанию.

Когда дочерний процесс создается с помощью **`fork(2)`**, он наследует копию своего родительского окружения.

По соглашению строки в окружении имеют форму **`"имя=значение"`**.

Имя чувствительно к регистру и не может содержать символ **`"="`**.

Значение может быть любым, что может быть представлено в виде строки.

Имя и значение не могут содержать встроенный нулевой байт (**`'\0'`**), так как предполагается, что он завершает строку.

Переменные среды могут быть помещены в среду оболочки командой **`export`** в **`sh(1)`** или командой **`setenv`**, если используется **`csh(1)`**.

Начальная среда оболочки заполняется различными способами, например определениями из **`/etc/environment`**.

Кроме того, различные сценарии инициализации оболочки, такие как общесистемный сценарий **/etc/profile** и сценарий инициализации для каждого пользователя, могут включать команды, добавляющие переменные в среду оболочки.

Для создания определений переменной среды в рамках процесса, выполняющего команду, оболочки в стиле Bourne (bash) поддерживают синтаксис:

```
NAME=value command
```

Команде может предшествовать несколько определений переменных, разделенных пробелом.

```
$ LC_MESSAGES=C /opt/slickeditor/bin/vse
$ rpm -qa | LANG=C sort
```

Аргументы также могут быть помещены в окружение в момент вызова **exec(3)**.

Программа на C может манипулировать ***своим*** окружением с помощью функций из **<stdlib.h> getenv(3)**, **putenv(3)**, **setenv(3)** и **unsetenv(3)**.

Переменные окружения, которые обычно встречаются в системе

Список неполный и включает только общие переменные, которые обычные пользователи видят в своей повседневной жизни.

Переменные среды, специфичные для конкретной программы или библиотечной функции, задокументированы в разделе **ENVIRONMENT** соответствующей страницы руководства.

USER — имя вошедшего в систему пользователя (используется некоторыми программами, производными от BSD).

LOGNAME — имя вошедшего в систему пользователя (используется некоторыми программами, производными от System-V). Устанавливается при входе в систему.

HOME — «домашний» каталог пользователя. Устанавливается при входе в систему.

LANG — имя, используемое для категорий локалей, если оно не переопределено переменной **LC_ALL** или более конкретными переменными среды, такими как **LC_COLLATE**, **LC_CTYPE**, **LC_MESSAGES**, **LC_MONETARY**, **LC_NUMERIC** и **LC_TIME** (информацию о переменных среды **LC_*** можно найти **locale(7)**).

PATH — последовательность префиксов каталогов, которую **sh(1)** и многие другие программы используют при поиске исполняемого файла, указанного как простое имя файла (т. е. путь, не содержащий в начале `'/'`). Префиксы разделяются двоеточиями `':'`.

Список префиксов просматривается от начала до конца путем проверки имени пути, образованного объединением префикса, косой черты и имени файла, до тех пор, пока не будет найден файл с разрешением на выполнение.

В качестве устаревшей функции префикс нулевой длины (указанный как два соседних двоеточия или начальное или завершающее двоеточие) интерпретируется как текущий рабочий каталог. Однако использование этой функции устарело, и в POSIX отмечено, что соответствующее приложение для указания текущего рабочего каталога должно использовать явное имя пути (например, **'.'**).

Аналогично тому, как используется **PATH**, некоторыми оболочками для поиска цели команды изменения каталога (cd <target>) используется **CDPATH**.

Для поиска справочных страниц man(1) использует **MANPATH**, и так далее.

PWD — текущий рабочий каталог. Устанавливается некоторыми оболочками.

SHELL — абсолютный путь к оболочке, используемой пользователем при входе. Устанавливается при входе в систему.

TERM — тип терминала, для которого должен быть подготовлен вывод.

PAGER — предпочитаемая пользователем утилита для отображения текстовых файлов. Это может быть любая строка, допустимая в качестве операнда командной строки для команды **sh -c**. Если **PAGER** имеет значение **null** или не установлен, то приложения, запускающие пейджер, по умолчанию будут использовать программу, такую как **less(1)** или **more(1)**.

EDITOR/VISUAL — предпочитаемая пользователем утилита для редактирования текстовых файлов. Это может быть любая строка, допустимая в качестве операнда командной строки для команды **sh -c**.

Наличие или значение определенных переменных среды влияет на поведение многих программ и библиотечных процедур:

Переменные **LANG**, **LANGUAGE**, **NLSPATH**, **LOCPATH**, **LC_ALL**, **LC_MESSAGES** и т. д. влияют на обработку локали (**catopen(3)**, **gettext(3)** и **locale(7)**).

TMPDIR влияет на префикс пути к имени, созданному **tempnam(3)** и другими подпрограммами, а также на временный каталог, используемый **sort(1)** и другими программами.

LD_LIBRARY_PATH, **LD_PRELOAD** и другие переменные **LD_*** влияют на поведение динамического загрузчика/компоновщика (**ld.so(8)**).

```
export LD_LIBRARY_PATH=$HOME/lib64
```

POSIXLY_CORRECT заставляет определенные программы и библиотеки следовать предписаниям POSIX.

На поведение **malloc(3)** влияют переменные **MALLOC_***.

TZ и **TZDIR** предоставляют информацию о часовом поясе, используемую **tzset(3)** и через нее такими функциями, как **ctime(3)**, **localtime(3)**, **mktime(3)**, **strftime(3)**, **tzselect(8)**.

PRINTER или **LPDEST** могут указать желаемый принтер для использования **lpr(1)**.

Переменная **HOSTALIASES** задает имя файла, содержащего псевдонимы, которые будут использоваться с **gethostbyname(3)**.

```
$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias mc='. /usr/libexec/mc/mc-wrapper.sh'
alias mingw32-env='eval `rpm --eval %{mingw32_env}`'
alias mingw64-env='eval `rpm --eval %{mingw64_env}`'
alias which='(alias; declare -f) | /usr/bin/which --tty-only --read-alias --
read-functions --show-tilde --show-dot'
alias xzegrep='xzegrep --color=auto'
alias xzfgrep='xzfgrep --color=auto'
alias xzgrep='xzgrep --color=auto'
alias zegrep='zegrep --color=auto'
alias zfgrep='zfgrep --color=auto'
alias zgrep='zgrep –color=auto'
```

TERMCAP дает информацию о том, как обращаться к данному терминалу (или предоставляет имя файла, содержащего такую информацию).

COLUMNS и **LINES** сообщают приложениям размер окна, возможно, переопределяя фактический размер.

Использование среды может иметь угрозу безопасности. Многие системные команды были обмануты пользователем, указанием необычных значений для **LD_LIBRARY_PATH**.

Существует также риск загрязнения пространства имен. Такие программы, как **make** и **autoconf**, позволяют переопределять имена утилит по умолчанию с помощью переменных с аналогичными именами во всех регистрах. Например, CC используется для выбора нужного компилятора C (и аналогично MAKE, AR, AS, FC, LD, LEX, RM, YACC и т. д.). Однако в некоторых традиционных случаях такая переменная среды дает опции для программы вместо имени пути. Такое использование считается ошибочным, и его следует избегать в новых программах.

Пример

parent.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/wait.h>

int main() {

    int child_status;
    char *args[] = {"pr0gramm", "parm_1", (char*)0};
    pid_t pid = fork();

    if (pid == -1) {
        printf("Error occured, error code - %d\n", errno);
        exit(errno);
    }
    if (pid == 0) {
        printf("Child process created. Please, wait...\n");
        execve("./child", args, NULL);
    }
    wait(&child_status);
    printf("Child process have ended with %d exit status\n", child_status);
    exit(0);
}
```

child.c

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

int main(int argc, char *argv[]) {

    fprintf(stdout, "Child process begins...\n");
    for (int i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
    exit(0);
}
```

makefile

```
CC = gcc
CFLAGS = -W -Wall -Wextra -std=c11
.PHONY: clean

all: parent child
parent: parent.c makefile
        $(CC) $(CFLAGS) parent.c -o parent
child: child.c makefile
        $(CC) $(CFLAGS) child.c -o child
clean:
        rm parent child
```

Компиляция и сборка

```
$ make
gcc -W -Wall -Wextra -std=c11 parent.c -o parent
gcc -W -Wall -Wextra -std=c11 child.c -o child
$ls -1
child
child.c
makefile
parent
parent.c
```

Запуск

```
$ ./parent
Parent process begins...
Child process created. Please, wait...
Child process begins...
pr0gramm
parm_1
Child process have ended with 0 exit status
```

wait, waitpid — ожидает завершения процесса

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Ожидает изменения состояния процесса-потомка и получения информации о потомке, чье состояние изменилось (завершение, останов/продолжение работы по сигналу).

Функция **wait** приостанавливает выполнение текущего процесса до тех пор, пока какой нибудь из дочерних процессов не завершится, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик.

Функция **waitpid** приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс, указанный в параметре **pid**, не завершит выполнение, или пока не появится сигнал, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик.

Если указанный дочерний процесс к моменту вызова функций уже завершился (так называемый «*зомби*» — "zombie"), то функция немедленно возвращает управление.

Системные ресурсы, связанные с дочерним процессом, освобождаются.

Если **status** не **NULL**, в нем будет сохранена информация о состоянии (см. ниже).

wait(&status) эквивалентна **waitpid(-1, &status, 0)**.

Параметр **pid** может принимать несколько значений: **-pid**, **-1**, **0**, **pid**

< -1 — означает, что нужно ждать любого дочернего процесса, идентификатор группы процессов которого равен абсолютному значению **pid**.

-1 — означает ожидание любого дочернего процесса (поведение функции **wait()**).

0 — означает ожидание любого дочернего процесса, идентификатор группы процессов которого равен идентификатору текущего процесса.

> 0 — означает ожидание дочернего процесса, чей идентификатор равен **pid**.

status — если при вызове **status** не равен **NULL**, то функции **wait** и **waitpid** сохраняют информацию о статусе в переменной, на которую указывает **status**.

Состояние **status** можно проверить с помощью макросов, которые принимают в качестве аргумента буфер (типа **int**), а не указатель на буфер!:

WIFEXITED(status) — не равно нулю, если дочерний процесс успешно завершился.

WEXITSTATUS(status) — возвращает восемь младших битов значения, которое вернул завершившийся дочерний процесс.

Эти биты могли быть установлены в аргументе функции **exit()** или в аргументе оператора **return** функции **main()**. Этот макрос можно использовать, только если **WIFEXITED** вернул ненулевое значение.

WIFSIGNALED(status) — возвращает истинное значение, если дочерний процесс завершился из-за необработанного сигнала.

WTERMSIG(status) — возвращает номер сигнала, который привел к завершению дочернего процесса. Этот макрос можно использовать, только если **WIFSIGNALED** вернул ненулевое значение.

WIFSTOPPED(status) — возвращает истинное значение, если дочерний процесс, из-за которого функция вернула управление, в настоящий момент остановлен; это возможно, только если использовался флаг **WUNTRACED** или когда подпроцесс отслеживается (см. **ptrace(2)**).

WSTOPSIG(status) — возвращает номер сигнала, из-за которого дочерний процесс был остановлен. Этот макрос можно использовать, только если **WIFSTOPPED** вернул ненулевое значение.

Некоторые версии Unix (например Linux, Solaris, но не AIX, SunOS) также определяют макрос **WCOREDUMP(status)** для проверки того, не вызвал ли дочерний процесс ошибку ядра.

Использовать его следует только в структуре

```
#ifdef WCOREDUMP
...
#endif
```

options — создается путем логического сложения нескольких следующих констант:

WNOHANG — означает немедленное возвращение управления, если ни один дочерний процесс не завершил выполнение.

WUNTRACED — означает возврат управления и для остановленных (но не отслеживаемых) дочерних процессов, о статусе которых еще не было сообщено. Состояние отслеживаемых остановленных подпроцессов обеспечивается и без этой опции.

Возвращаемые значения

-1 в случае ошибки (в этом случае переменной **errno** присваивается соответствующее значение).

Идентификатор дочернего процесса, который завершил выполнение, или ноль, если использовался **WNOHANG** и ни один дочерний процесс пока еще недоступен.

Ошибки

ECHILD — процесс, указанный в **pid**, не существует или не является дочерним процессом текущего процесса. (Это может случиться и с собственным дочерним процессом, если обработчик сигнала **SIGCHLD** установлен **в SIG_IGN**. См. главу ЗАМЕЧАНИЯ по поводу многозадачности процессов.)

EINVAL — аргумент **options** неверен.

EINTR — использовался флаг **WNOHANG**, и был получен необработанный сигнал или **SIGCHLD**. Стандарт Single Unix Specification описывает флаг **SA_NOCLDWAIT** (не поддерживается в Linux), если он установлен или обработчик сигнала **SIGCHLD** устанавливается в **SIG_IGN**, то завершившиеся дочерние процессы не становятся зомби, а вызов **wait()** или **waitpid()** блокируется, пока все дочерние процессы не завершатся, а затем устанавливает переменную **errno** равной **ECHILD**.

Стандарт POSIX оставляет неопределенным поведение при установке **SIGCHLD** в **SIG_IGN**. поздние стандарты, включая SUSv2 и POSIX 1003.1-2001, определяют поведение, только что описанное как опция совместимости с XSI.

Linux не следует второму варианту — если вызов **wait()** или **waitpid()** сделан в то время, когда **SIGCHLD** игнорируется, то вызов ведет себя, как если бы **SIGCHLD** не игнорировался, то есть вызов блокирует до завершения работы следующего подпроцесса и возврата идентификатора процесса PID и статуса этого подпроцесса.

Детали в man 2 wait.

_exit — функция, завершающая работу программы

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

void _Exit(int status);
```

_exit "немедленно" завершает работу программы.

При этом:

- все дескрипторы файлов, принадлежащие процессу, закрываются;

- все его дочерние процессы начинают управляться процессом 1 (init/systemd), а родительскому процессу посылается сигнал **SIGCHLD**;

- значение **status** возвращается родительскому процессу как статус завершаемого процесса (он может быть получен с помощью одной из функций семейства **wait)**.

Функция **_Exit** эквивалентна функции **_exit**.

Возвращаемые значения

Эти функции никогда не возвращают управление вызвавшей их программе.

Соответствие стандартам

SVr4, SVID, POSIX, X/OPEN, BSD 4.3. Функция **_Exit()** была представлена в C99.

Замечания

Для рассмотрения эффектов завершения работы, передачу статуса выхода, зомби-процессов, сигналов и т.п., следует смотреть документацию по **exit(3)**.

Функция **exit()** выполняет нормальное завершение процесса и возвращает значение **status & 0377** породившему процессу.

Функция **_exit(2)** аналогична **exit(3)**, но не вызывает никаких функций, зарегистрированных с С-функцией **atexit(3)**, а также не вызывает никаких зарегистрированных обработчиков сигналов.

С-функция **exit(3)** выполняет сброс стандартных буферов ввода-вывода и удаление временных файлов, созданных **tmpfile(3)**, в отличие от этого поведение **_exit(2)** зависит от реализации.

С другой стороны, **_exit** закрывает открытые дескрипторы файлов, а это может привести к неопределенной задержке для завершения вывода данных. Если задержка нежелательна, то может быть полезным перед вызовом **_exit()** вызывать функции типа **tcflush()**.

Опять же, будет ли завершен ввод-вывод, а также какие именно операции ввода-вывода будут завершены при вызове **_exit()**, зависит от реализации.