



ЛАБ 1



Задание

- **-i** Печатает реальные и эффективные идентификаторы пользователя и группы.
- **-s** Процесс становится лидером группы. *Подсказка:* смотри **setpgid(2)**.
- **-p** Печатает идентификаторы процесса, процесса-родителя и группы процессов.
- **-u** Печатает значение **ulimit**
- **-Unew_ulimit** Изменяет значение **ulimit**. *Подсказка:* смотри **atol(3С)** на странице руководства **strtol(3С)**
- **-c** Печатает размер в байтах core-файла, который может быть создан.
- **-Csize** Изменяет размер core-файла
- **-d** Печатает текущую рабочую директорию
- **-v** Распечатывает переменные среды и их значения
- **-Vname=value** Вносит новую переменную в среду или изменяет значение существующей переменной.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <ulimit.h>
#include <sys/resource.h>
#include <limits.h>
#include <sys/param.h>

extern char **environ;

void main(int argc, char *argv[]) {
    char option;
    struct rlimit rlim;

    if (argc < 2) {
        perror("Add an option");
        exit(EXIT_FAILURE);
    }

    while ((option = getopt(argc, argv, "ispuU:cC:dvV:")) != EOF) {
        switch (option) {
        case 'i':
            printf("User id = %d\n", getuid());
            printf("Effective user id = %d\n", geteuid());
            printf("Group id = %d\n", getgid());
            printf("Effective group id = %d\n", getegid());
            break;

        case 's':
            if (setpgid(0, 0) == -1){
                perror("Cannot set a leader of group\n");
                exit(EXIT_FAILURE);
            }
            else
                printf("Now process is a leader of the group\n");
            break;

        case 'p':
            printf("Process id = %d\n", getpid());
            printf("Parent id = %d\n", getppid());
            printf("Group id = %d\n", getpgid(0));
            break;

        case 'u':
            printf("Ulimit value is: %lu bytes\n", ulimit(UL_GETFSIZE));
            break;

        case 'U':
            if(atol(optarg) > ulimit(UL_GETFSIZE))
```

```

        perror("Not enough rights to change limit\n");
    else {
        ulimit(UL_SETFSIZE, atol(optarg));
        printf("Ulimit was successfully changed\n");
    }
    break;

case 'c': {
    getrlimit(RLIMIT_CORE, &rlim);
    printf("Soft limit of core dump is: %lu bytes\n", rlim.rlim_cur);
    printf("Hard limit of core dump is: %lu bytes\n", rlim.rlim_max);
    printf("RLIM_INFINITY is: %lu\n", RLIM_INFINITY);
    break;
}

case 'C': {
    struct rlimit rlim1;
    getrlimit(RLIMIT_CORE, &rlim1);
    rlim1.rlim_cur = atoll(optarg);

    if(rlim1.rlim_cur > rlim1.rlim_max){
        perror("Current soft limit is bigger than a hard limit\n");
        exit(EXIT_FAILURE);
    }
    else {
        setrlimit(RLIMIT_CORE, &rlim1);
        printf("Soft limit of core file was successfully changed\n");
    }
    break;
}

case 'd': {
    char* cwd;
    long maxLen = pathconf("/", _PC_PATH_MAX);
    if((cwd = getcwd(NULL, maxLen)) == NULL){
        perror("Finding current working directory failed");
        exit(EXIT_FAILURE);
    }
    printf("Current directory is: %s\n", cwd);
    free(cwd);
    break;
}

case 'v':
    printf("Environment variables:\n");
    for (char **p = environ; *p != 0; p++)
        printf("%s\n", *p);
    break;

case 'V':
    if (putenv(optarg) == -1){
        perror("Check the format of variable\n");
        exit(EXIT_FAILURE);
    }
    else
        printf("Environment variable was successfully added\n");
    break;

default:
    perror("Check the option for correctness\n");
    break;
}
}
}

```

▼ getopt()

```
getopt(int argc, char *const argv[], const char *optstring)
```

getopt(3С) - это функция общего назначения для обработки опций командной строки.

Командная строка должна удовлетворять следующим правилам:

Общий формат: имя_команды [опции] [другие аргументы]

имя_команды Имя выполняемого файла

`getopt(3C)` использует четыре внешних переменных(объявлены в `<unistd.h>`): // man

- **optarg** указатель на символ. Когда `getopt(3C)` обрабатывает опцию, у которой есть аргументы, optarg содержит адрес этого аргумента.

▼ CASE 'U': Реальный и эффективный ID пользователя/группы

При входе в систему идентификаторы пользователя и группы устанавливаются из файла `/etc/passwd`. Реальные и эффективные идентификаторы для процесса первоначально совпадают. Эффективный идентификатор пользователя и список групп доступа (показываемый `getgroups(2)`) используются для определения прав доступа к файлам.

Владелец любого файла, созданного процессом, определяется эффективным идентификатором пользователя, а группа файла - эффективным идентификатором группы.

В Unix, исполняемые файлы могут иметь специальные атрибуты: биты установки идентификатора пользователя или группы (`setuid`- и `setgid`-биты). Эти биты устанавливаются с помощью команды `chmod(1)` или системного вызова `chmod(2)`. Биты `setuid` и `setgid` показываются командой `ls -l` буквой 's' на месте обычного расположения 'x', например:

```
$ ls -l `which su`  
-r-sr-xr-x 1 root sys 38656 Oct 21 2011 /usr/bin/su
```

Если один или оба этих бита установлены, при запуске такого файла, эффективный идентификатор пользователя и/или группы у процесса становится таким же, как и у владельца и/или группы файла с программой. Благодаря этому механизму можно стать «заместителем» или «представителем» привилегированной группы.

Используя программы с `setuid`-битом, можно получить доступ к файлам и устройствам, которые обычным образом недостижимы. Например, если какой-либо файл данных доступен по чтению и записи только для владельца, другие пользователи не могут получить доступ к этому файлу. Если же владелец этого файла напишет программу доступа к этому файлу и установит `setuid`-бит, тогда все пользователи данной программы смогут получить доступ к файлу, ранее недостижимому. Патент на механизм установки идентификатора пользователя

▼ CASE 'S': Лидер группы процессов

Создается группа с идентификатором = идентификатору переводимого процесса, состоящая изначально только из этого процесса

`SETPID()`

Системный вызов `setpgid` служит для перевода процесса из одной группы процессов в другую, а также для создания новой группы процессов.

Параметр `pid` является идентификатором процесса, который нужно перевести в другую группу, а параметр `pgid` – идентификатором группы процессов, в которую предстоит перевести этот процесс.

Не все комбинации этих параметров разрешены. Перевести в другую группу процесс может либо самого себя (и то не во всякую, и не всегда), либо свой процесс-ребенок, который не выполнял системный вызов `exec()`, т.е. не запускал на выполнение другую программу.

- Если параметр `pid` равен `0`, то считается, что процесс переводит в другую группу самого себя.
- Если параметр `pgid` равен `0`, то в Linux считается, что процесс переводится в группу с идентификатором, совпадающим с идентификатором процесса, определяемого первым параметром.
- Если значения, определяемые параметрами `pid` и `pgid`, равны, то создается новая группа с идентификатором, совпадающим с идентификатором переводимого процесса, состоящая первоначально только из этого процесса. **Переход в другую группу без создания новой группы возможен только в пределах одного сеанса.**

В новую группу не может перейти процесс, являющийся **лидером группы**, т.е. процесс, идентификатор которого совпадает с идентификатором его группы.

Лидер группы процессов



Лидер группы процессов — это процесс, чей `pid` совпадает с `pgid` группы.

Правила применения `setpgid()` несколько сложны.

1. Процесс может устанавливать группу для себя или одного из своих потомков. Он не может изменять группу для любого другого процесса в системе, даже если процесс, вызвавший `setpgid()`, имеет административные полномочия.
2. Лидер сеанса не может изменить свою группу.
3. Процесс не может быть перемещен в группу, чей лидер представляет другой сеанс, чем он сам. Другими словами, все процессы в группе должны относиться к одному и тому же сеансу.

Лидер сеанса

Все процессы объединены в сессии. Процессы, принадлежащие к одной сессии, определяются общим идентификатором сессии (`sid`). Лидер сессии – это процесс, который создал сессию вызовом `setsid(2)`. Идентификатор процесса лидера сессии совпадает с его `sid`. Сессия может выделить себе управляющий терминал для того, чтобы дать пользователю возможность управлять исполнением заданий (групп процессов) внутри сессии. При входе в систему создается сессия, которая имеет идентификатор сессии, равный идентификатору процесса вашего входного shell'a. Также, при открытии каждого окна `xterm(1)` или закладки

▼ CASE 'P': ID процесса, родительский процесс, группа процессов

. Идентификатор процесса:

- `getpid(2)` возвращает идентификатор процесса. Например: `pid=getpid();`
- `getppid(2)` возвращает идентификатор родительского процесса. Например: `ppid=getppid();`
- `getpgid(2)` возвращает идентификатор группы для процесса, идентификатор которого равен `pid`, или для вызывающего процесса, если `pid` равен 0. Например: `pgid=getpgid(0);` Замечание: группы процессов обсуждаются позже в этом курсе.

ИДЕНТИФИКАТОР ПРОЦЕССА ID

Каждый процесс в Unix имеет идентификатор процесса (process id, **pid**). Строго говоря, этот идентификатор не является уникальным: система переиспользует свободные значения идентификаторов при создании новых процессов. Однако, одновременно в системе не может быть двух процессов с одинаковыми pid.

ГРУППА ПРОЦЕССОВ

Группа процессов — это коллекция из одного или более процессов, обычно связанных с выполнением одного и того же задания (управление заданиями рассматривается в разделе 9.8), которые могут принимать сигналы от одного и того же терминала. Каждая группа процессов имеет уникальный идентификатор. Идентификатор группы процессов напоминает идентификатор процесса: это целое положительное число, которое может храниться в переменной типа **pid_t**. Функция

▼ CASE ‘u’: Вывод значения ulimit

- Ядро ОС может накладывать на процессы ограничения количественные, при которых процесс вправе использовать только те или иные ресурсы системы, но только до определенного предела.
- Вызов аналогичен `setrlimit(RLIMIT_FSIZE)`, cmd определена в `<ulimit.h>`

Для каждого вида количественного ограничения система позволяет установить два предельных уровня — мягкий и жёсткий (*soft/hard*). Ядро в своих проверках использует мягкий уровень, но любой процесс может изменить этот уровень (как уменьшить, так и увеличить) для любого из своих ограничений в пределах от нуля до установленного жёсткого уровня. Что касается жёсткого уровня, то любой процесс может его уменьшить, но увеличивать его разрешается только процессам, работающим с правами суперпользователя; для всех прочих процессов уменьшение жёсткого уровня любого из лимитов необратимо.

«жестким» пределами. Мягкий (административный) предел устанавливается `setrlimit(2)` с командой `RLIMIT_NOFILE` или командой `ulimit(1)`. Жёсткий предел устанавливается настройками ядра системы. Значение жёсткого предела можно определить системным

Системный вызов `ulimit(2)` используется, чтобы определять и устанавливать некоторые ограничения. Этот вызов аналогичен системным вызовам `getrlimit(2)/setrlimit(2)`, но появился гораздо раньше и поддерживается для совместимости со старыми программами. Рекомендуется использовать `getrlimit(2)/setrlimit(2)`. Встроенная команда shell `ulimit(1)` также может использоваться для установки этих ограничений.

Аргумент `cmd` может принять значение одной из следующих символьных констант:

UL_GETFSIZE Возвращает текущее ограничение процесса на максимальный размер файла. Размер измеряется в блоках по 512 байт.

▼ CASE ‘U’: Изменить значение Ulimit (Подсказка: смотри [atol\(3C\)](#))

- `ulimit` - get and set process limits
- Аналогична вызову `setrlimit(RLIMIT_FSIZE)`

ULIMIT

ОПИСАНИЕ

Системный вызов `ulimit` позволяет управлять ограничениями, наложенными на процесс. Аргумент `cmd` может принимать следующие значения:

символьных констант

UL_SETFSIZE Устанавливает ограничение на размер файла. Только суперпользователь^в может увеличить это ограничение. Остальные могут только уменьшить его. Может быть полезно ограничить размер файла при отладке программы, которая создает файлы или удлиняет существующие файлы.

`newlimit` используется при `cmd` равном `UL_SETFSIZE`. Это новый размер файла в блоках.

ATOL

НАЗВАНИЕ

`strtol`, `atol`, `atoi` - преобразование цепочки символов в целое число

`Atol (str)` эквивалентно `strtol (str, (char **) NULL, 10)`.

```
long strtol (str, ptr, base)
char *str, **ptr;
int base;

long atol (str)
char *str;

int atoi (str)
char *str;
```

Результатом функции `strtol` является целое число типа `long`, заданное цепочкой символов, на которую указывает аргумент `str`. Цепочка просматривается до первого символа, несовместимого с выбранной системой счисления. Начальные пробельные символы [см. макрос `isspace` в [ctype\(3C\)](#)] игнорируются.

Если аргумент `base` положителен (и не превосходит 36), то он рассматривается как основание системы счисления.

Если аргумент `ptr` не равен `(char **) NULL`, то в слове, на которое он ссылается, возвращается указатель на символ, вызвавший завершение просмотра. Если число сформировать не удается, то `*ptr` устанавливается равным `str` и в качестве результата возвращается ноль.

OPTARG - содержит адрес аргумента опции от функции getopt(3C)

▼ CASE ‘c’: Размер core-файла

CORE ФАЙЛ

²⁵Core-файл — это файл с именем `core` или `prog.core` (где `prog` — имя программы), создаваемый операционной системой в текущем каталоге при аварийном завершении процесса. В этот файл полностью записывается содержимое сегментов данных и стека на момент аварии. Core-файлы позволяют с помощью отладчика проанализировать причины аварии, в том числе узнать точку кода, в которой произошла авария, посмотреть значения переменных на момент аварии и т. д.; мы рассмотрим работу с ними в приложении,

Как уже упоминалось, сигналы SIGABRT, SIGBUS, SIGSEGV, SIGQUIT, SIGILL, SIGTRAP, SIGSYS, SIGXCPU, SIGXFSZ и SIGFPE приводят к аварийному завершению и обычно сопровождаются сбросом образа памяти на диск. Образ памяти процесса записывается в файл с именем `core` в текущем рабочем каталоге процесса (термин `core`, или *сердечник*, напоминает о временах, когда оперативная память состояла из матриц ферритовых сердечников). Файл `core` будет содержать значения всех переменных программы, регистров процессора и необходимую управляющую информацию ядра на момент завершения программы. Статус завершения процесса после аварийного завершения будет тем же, каким он был бы в случае нормального завершения из-за этого сигнала, только в нем будет дополнительно выставлен седьмой бит младшего байта. В большинстве создаётся в текущей рабочей директории

RLIMIT_CORE

- Максимальный размер core-file (Если = 0, то файл не создается, Если > 0, то файлы обрезаются до этого размера)
- Значение RLIMIT_CORE может варьироваться от процесса к процессу, также в зависимости от конфигурации системы
- `RLIMIT_CORE` — максимальный возможный размер core-файла²⁵, создаваемого при авариях (в байтах); если установить нулевой лимит, core-файлы создаваться не будут;

GETRLIMIT() - Системный вызов для доступа к среде исполнения процесса

Любой процесс имеет ряд ограничений на использование ресурсов. Некоторые из этих ограничений можно изменить с помощью функций `getrlimit` и `setrlimit`.

```
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlptr);

int setrlimit(int resource, const struct rlimit *rlptr);
```

Обе возвращают 0 в случае успеха,
–1 – в случае ошибки

Эти две функции определены стандартом *Single UNIX Specification* как расширения XSI. Ограничения на ресурсы для процесса обычно устанавливаются процессом с идентификатором 0 во время инициализации системы и затем наследуются остальными процессами. Каждая реализация предлагает собственный способ настройки различных ограничений.

При обращении к этим функциям им передается ресурс (*resource*) и указатель на следующую структуру:

```
struct rlimit {
    rlim_t rlim_cur; /* мягкий предел: текущий предел */
    rlim_t rlim_max; /* жесткий предел: максимальное значение для rlim_cur */
};
```

RLIM_INFINITY

В Linux `RLIMIT_CORE` ограничение (см. `setrlimit(2)` справочную страницу) выражается в байтах. `ulimit` В то время как для `bash` большинства других оболочек это выражается в кибибайтах.

Максимальное значение `RLIMIT_CORE` в байтах, которое вы можете передать в `setrlimit()` Linux, будет наибольшим 64-битным целым числом без знака (18446744073709551615), принимая во внимание, что на самом деле это специальное значение для `RLIM_INFINITY`, максимальное значение, которое вы можете передать в `bash`, `ulimit` будет разделено на 1024, 18014398509481983, но тогда это не установит ограничение на `RLIMIT_INFINITY` 18446744073709550592 18014398509481983 * 1024, что на 1023 меньше `RLIM_INFINITY`.

РЕСУРСЫ

МЯГКИЙ И ЖЕСТКИЙ ПРЕДЕЛ

«жестким» пределами. **Мягкий** (административный) предел устанавливается `setrlimit(2)` с командой `RLIMIT_NOFILE` или командой `ulimit(1)`. Жёсткий предел устанавливается настройками ядра системы. Значение жёсткого предела можно определить системным вызовом `sysconf(2)` с параметром `_SC_OPEN_MAX`. В Solaris, жесткий предел устанавливается параметром `rlim_fd_max` в файле `/etc/system` (`system(4)`); его изменение требует административных привилегий и перезагрузки системы.

В Solaris 10, максимальное значение `rlim_fd_max` равно `MAXINT` (2147483647).

▼ CASE ‘C’ - Изменить значение core-file

SETRLIMIT()

Поменяли мягкое ограничение ресурса в структуре на введенное значение

- `setrlimit(2)` устанавливает некоторые программные и аппаратные ограничения процесса.

РЕСУРСЫ

Изменение пределов ресурсов производится в соответствии со следующими тремя правилами.

1. Процесс может изменять значение мягкого предела при условии, что оно не превышает жесткий предел.
2. Процесс может понизить значение жесткого предела вплоть до значения мягкого предела. Операция понижения жесткого предела необратима для рядовых пользователей.
3. Только процесс, обладающий привилегиями суперпользователя, может поднять значение жесткого предела.

▼ CASE ‘d’: Текущая рабочая директория

GETCWD()

Функция `getcwd()` копирует полный путь (максимум `len` символов) текущего рабочего каталога диска в строку, на которую указывает параметр `dir`. Если полный путь длиннее, чем `len` символов, то возникает ошибка. Функция `getcwd()` возвращает указатель на `dir`.

При вызове функции `getcwd()` с параметром `dir`, равным `NULL`, функция `getcwd()` автоматически размещает буфер с использованием функции `malloc()` и возвращает указатель на этот буфер. Можно освободить память, выделенную функцией `getcwd()`, используя функцию `free()`.

ДЛИНА СТРОКИ

Некоторые параметры, например, максимальная длина имени файла, зависят от файловой системы.

В старых учебниках, для проверки значения этих параметров рекомендовалось использовать препроцессорные макросы, определенные в заголовочном файле `<limits.h>`. В Solaris этот файл генерируется автоматически при генерации системы, и значения в нем соответствуют значениям, использованным при компиляции ядра системы. Если программа исполняется на другой версии ядра, значения некоторых параметров могут измениться.

PATHCONF: `long pathconf(char *path, int name);`

`fd` — дескриптор открытого файла (это понятие будет обсуждаться в разделе «Файловый ввод-вывод»). Параметры будут определены для файловой системы, в которой размещен этот файл.

`path` — путевое имя файла или каталога, определяющее файловую систему, для которой необходимо определить параметр.

`name` — имя параметра. В качестве имен рекомендуется использовать символьные константы, определенные в `<unistd.h>`. Полный список имен приведен на странице руководства `pathconf(2)`.

Пример — определение максимальной допустимой длины путевого имени файла:

```
long val;  
val=pathconf("/", _PC_PATH_MAX);
```

▼ CASE 'v': Печать переменных окружения

ENVIRON

Стандарт ISO C определяет только два аргумента функции `main`, а передача среды окружения через третий аргумент не дает никаких преимуществ перед передачей той же информации через глобальную переменную `environ`. Поэтому стандарт POSIX.1 указывает, что передача среды окружения должна осуществляться не через третий аргумент функции `main`, а через глобальную переменную `environ` (если это возможно). Доступ к конкретным переменным окружения обычно осуществляется с помощью функций `getenv` и `putenv`, которые будут описаны в разделе 7.9, а не через глобальную переменную `environ`. Однако для просмотра всех переменных окружения следует использовать указатель `environ`.

Глобальная переменная `environ` указывает на массив строк, называемый окружением (массив указателей на строки) Эти строки, которые заканчиваются нулевым символом, и есть **переменные окружения**, представленные в виде “имя=значение”

В самом начале: переменные окружения лежат изначально в конфигурационных файлах(/etc - для хранения системных параметров, настроек служб и приложений)

1. **Init** — это процесс, порождающий все другие пользовательские процессы. Он читает командный файл `/etc/inittab` (см. `inittab(4)`) и запускает все остальные задачи в системе, используя **fork(2)** и **exec(2)**. `inittab` определяет, какие процессы должны быть порождены на конкретном уровне запуска системы

2. В многопользовательском режиме **init** запускает `ttymon` для консоли, `ttymon` устанавливает переменную `TERM` и

определяет активность терминального порта и выдает приглашение входа в систему. После ответа на приглашение входа в систему, ttymon запускает программу /bin/login.

3. Программа .login проверяет регистрационное имя пользователя и его пароль. При успешной проверке, login запускает поверх себя командный интерпретатор shell, определенный в файле passwd. login производит установку следующих переменных среды исполнения.

- потом передаются в процесс init и у него они хранятся в стеке, init их передает (либо вызовом exec меняет) в дочерние процессы которые могут этот список под себя менять

Переменные окружения изначально лежат в пользовательском стеке процесса на его дне, затем сверху добавляются новые переменные. Так же они могут лежать в динамических сегментах - кладутся туда с помощью **Putenv()**. Putenv, в свою очередь, использует malloc для выделения памяти под новые переменные.

▼ CASE'V : Изменить / Добавить переменную окружения

PUTENV(1)

- Указатель, передаваемый в функцию `putenv()`, всегда должен указывать **на статические данные** (не автоматические переменные). В C и C++ строковые константы размещены в статическом сегменте и постоянны во время всей жизни программы

SETENV(3)

`int setenv(const char* name, const char* value, int overwrite)` - устанавливает новое значение переменной:

- Если переменной не было - значение устанавливается в любом случае
- Если была - новое значение устанавливается при overwrite = 0, те overwrite разрешает или запрещает менять значение ПО

При добавлении новых переменных, putenv/setenv могут выделить новую память для размещения массива envp при помощи malloc(3C). При этом изменится значение переменной envp, но НЕ третьего параметра main.

▼ ОТЛИЧИЕ SETENV() ОТ PUTENV()

Setenv - если параметр overwrite не 0 ⇒ переменная заменяется.

Функция `int setenv(const char* name, const char* value, int overwrite)` - устанавливает новое значение переменной:

- Если переменной не было - значение устанавливается в любом случае
- Если была - новое значение устанавливается при overwrite = 0, те overwrite запрещает менять ПО
- You must not pass an auto array to it
- If you modify the string, you modify the environment
- After removing the name from the environment you can free the string and therefore not leak memory

```
t setenv(const char *envname,
          const char *envval,
          int overwrite);
```

env takes the name and value separately and allows you to back out of the variable if it is already set. This is adequate for the following cases:

- You can pass an auto array
- You can modify the values you passed without affecting the environment
- You have no idea (in general) what, if anything, to free when you remove the value from the environment

▼ СЕГМЕНТЫ ВАП

Когда программа загружается в память, она становится активным процессом, которому как-бы доступно все адресное пространство машины (так как оно не разделено на сегменты).

Виртуальная память пользователя для каждого процесса подразделяется на три сегмента: **текст, данные и пользовательский стек**. Когда программа загружается в память, информация из файла a.out используется для размещения и инициализации сегментов текста и данных.

Сегмент текста содержит команды, полученные из кода, написанного пользователем. Текст всегда загружается по одному и тому же виртуальному адресу. Это виртуальный адрес зависит от типа машины и для многих компьютеров равен нулю.

За сегментом текста следует **сегмент данных**, содержащий статические и внешние переменные.

Стек пользователя используется для сохранения активационных записей, содержащих локальные переменные, аргументы вызовов функций, значения регистров, возвращаемые значения функций и другую информацию. При любом вызове функции запись активации помещается в стек. При возвращении из функции запись активации извлекается из стека. Место расположения стека и направление его роста машинно-зависимы. Любая попытка обращения к памяти между окончанием сегмента данных и вершиной стека приводит к ошибке обращения к памяти.

▼ ММАР()



mmap() - отображение файла на память. Файлы обычно лежат на диске, диск это не память!

putenv() - The putenv() function uses **malloc** (3C) to enlarge the environment.

malloc() - malloc не является системным вызовом, он предоставляется библиотекой C. реализация malloc в Glibc либо получает память из системного вызова brk()/sbrk(), либо анонимную память через mmap().

- Меньшие блоки (<128к): реализовано с помощью схемы buddy.
- Большой блок (> 128 КБ): создает анонимное сопоставление памяти. анонимное отображение памяти — это просто большой, заполненный нулями блок памяти, готовый к использованию.

Система устанавливает в дескрипторах всех страниц отображённой памяти **бит отсутствия**. При первом обращении к такой странице, диспетчер памяти генерирует исключение отсутствия страницы. Ядро системы перехватывает это исключение, считывает страницу из файла или с устройства, снимает бит отсутствия в дескрипторе страницы и возвращает управление программе. Для пользовательской программы это выглядит так, как будто прочитанные с устройства данные всегда находились в странице, на которую они отображены. **Таким образом можно отображать на память не только регулярные файлы, но и устройства, поддерживающие функцию lseek(2)**.

Для отображения файла на память, файл должен по-прежнему открываться вызовом open(2) и закрываться вызовом close(2).

После отображения функцией mmap(2), к содержимому файла можно обращаться, как к оперативной памяти.

1. Есть таблица трансляции(страниц), которой оперирует ДП. Таблица является частью контекста процесса. Хранится в оперативной памяти. На некоторых машинах таблица страниц должна быть загружена (операционной системой) в аппаратные регистры при каждом переключении центрального процессора с одного процесса на другой

Хранение:

Для решения проблемы хранения таблицы страниц используется **ассоциативная память - TLB** - **Зачастую это устройство находится внутри диспетчера памяти и состоит из небольшого количества записей**, их количество редко превышает 64.

Когда TLB загружается из таблицы страниц, все поля берутся из памяти \Rightarrow таблица страниц хранится в памяти

Номер виртуального адреса - индекс в таблице трансляции. Из записи в таблице страниц берется номер страничного блока (если таковой имеется). Номер страничного блока присоединяется к старшим битам смещения, заменяя собой номер виртуальной страницы, чтобы сформировать физический адрес, который может быть послан к памяти.

2. При обращении к странице ДП проверяет бит присутствия - если он = 0 \Rightarrow генерирует исключение отсутствия страницы / ошибку диспетчера памяти - **аппаратное исключение**.



Прерывания и исключения - программные решения для взаимодействия с аппаратными.

- **Прерывания внешние/аппаратные** (нужны для реализации переключения по инициативе ОС) - процессор получает внешний сигнал, останавливает то, что он делал и начинает делать что-то другое. Позволяют передать управление в какую-то точку, которая лежит в векторе прерываний (те вне контроллера программы)
 - **Прерывания внутренние/программные** (команда, триггерится самой командой. Чаще для реализации системных вызовов)
- Например, BIOS
- **Аппаратные исключения** - события в самом процессоре, вызванные им, когда процессор видит команду, которую не может исполнить (деление на 0). INTEL прекращает делать текущую команду, зовет обработчик аппаратного исключения (она может убить сигналом процесс)
 - **Программные исключения** - в Java, C++ - ошибки, вызывающие падение программы

РАЗНИЦА:

- Аппаратные прерывания - процессор обязан доделать текущую команду.
Остальные - процессор не может ее доделать (В случае диспетчера памяти - страница подкачивается и работа продолжается)
- Аппаратные исключения: Счетчик команд указывает на текущую команду
Остальные : счетчик указывает на следующую

3. Ищется свободный страничный блок, если свободного нет, то с помощью какого-то из алгоритма замещения страниц находится блок и сбрасывается на диск, если была модифицирована
4. Каждому отдельному исключению и каждому прерыванию, требующему специальной обработки процессором, назначается уникальный идентификатор - **вектор прерываний**. У Интела под исключения зарезервированы номера 1-31

Процессор сохраняет часть своего состояния (обычно 2-3 состояния):

- Если процессор без защиты памяти, то регистр состояния процессора и счетчик команд.
- Если система с защитой памяти - у таких процессоров 2 стека. Прерывания почти всегда обрабатываются в системном уровне привилегий, надо переключиться на системный стек где и сохраняются стек, счетчик команд и состояние.

У процессора есть табличка (вектор прерываний - либо таблица, либо записи в ней) № вектора - индекс этой таблицы. Эта запись - новое значение счетчика команд и состояния процессора.



По таблице вызывается **обработчик прерываний** - процедура в ядре (в случае системы с защищенной памятью), либо просто программа.

Обработчик прерываний ДП загружает нужные страницы с диска в оперативную память.

bss(неинициализированные переменные) - не из файла с флагом anon ~ dev/zero - Не обязаны указывать дескриптор открытого файла и смещение:

Отображение псевдоустройства /dev/zero выделяет вызывающей программе заполненный нулями блок виртуальной памяти указанного размера (Делается драйвером псевдоустройства dev/zero) - Выделяется страница, заполненная 0.

4. По завершении работы - обработчик возвращает текущее состояние процессора, слово состояния процессора, счетчик команд
5. Обновляется таблица страниц и TLB
6. Продолжается работа с незавершенной команды



1. Счетчик команд, слово состояния программы, а иногда и один или несколько регистров помещаются в текущий стек аппаратными средствами прерывания.

2. Затем компьютер переходит на адрес, указанный в векторе прерывания. На этом работа аппаратных средств заканчивается и вступает в действие программное обеспечение, а именно процедура обслуживания прерывания.

3. Все прерывания сначала сохраняют состояния регистров, зачастую используя для этого запись текущего процесса в таблице процессов. Затем информация, помещенная в стек прерыванием, удаляется и указатель стека переустанавливается на временный стек, используемый обработчиком прерывания. Такие действия выполняются подпрограммой на языке ассемблера.

4. Когда эта подпрограмма завершает свою работу, она вызывает С-процедуру, которая делает всю остальную работу для данного конкретного типа прерывания.

5. Возможно, когда работа этой процедуры будет завершена, какой-нибудь процесс переходит в состояние готовности к работе и вызывается планировщик, чтобы определить, какой процесс будет выполняться следующим.

6. После этого управление передается обратно коду, написанному на языке ассемблера, чтобы он загрузил для нового текущего процесса регистры и карту памяти и запустил выполнение этого процесса



Счетчик команд - адрес ячейки памяти со следующей выбираемой командой. После выборки этой команды счетчик команд обновляется, переставляя указатель на следующую команду.

Вектор прерываний - область памяти (обычно это фиксированная область в нижних адресах), связанная с каждым классом устройств ввода-вывода.

Вектор прерываний - номер устройства может быть использован как индекс части памяти, используемой для поиска адреса обработчика прерываний данного устройства.

Указатель стека - ссылается на вершину текущего стека в памяти. Стек содержит по одному фрейму (области данных) для каждой процедуры, в которую уже вошла, но из которой еще не вышла программа. В стековом фрейме процедуры хранятся ее входные параметры, а также локальные и временные переменные, не содержащиеся в регистрах.

Слово состояния процесса - в регистре содержатся биты кода условия, устанавливаемые инструкциями сравнения, а также биты управления приоритетом центрального процессора, режимом (пользовательским или ядра) и другие служебные биты.

Обычно режимом(системный или пользовательский) управляет специальный бит в слове состояния программы, его нельзя менять в пользовательском режиме

Обычно пользовательские программы могут считывать весь регистр целиком, но записывать — только в некоторые из его полей.

Контроллер - микросхема или набор микросхем, которые управляют устройством на физическом уровне. Он принимает от операционной системы команды, например считать данные с помощью устройства, а затем их выполняет.

Задачей контроллера является предоставление операционной системе простого (но не упрощенного) интерфейса

Обработчик прерываний - часть драйвера устройства, вызывающего прерывания

▼ PERROR()

ОПИСАНИЕ работа функции. `perror(3C)` печатает текст, переданный в качестве аргумента, затем двоеточие и пробел, затем сообщение об ошибке по текущему значению `errno` и перевод строки. Описание сообщений об ошибках можно найти на странице Руководства

Сообщения об ошибках должны выдаваться в `stderr`, содержать информацию о контексте ошибки и системное сообщение, соответствующее значению `errno`. Рекомендуется использовать `perror`.