OS Lab 1

Status	approved
checkbox	☑
class	OS
due date	@Feb 6, 2021

Task

Вывод различных атрибутов процесса в соответствии с указанными опциями

Напишите программу, которая будет обрабатывать опции, приведенные ниже. Опции должны быть обработаны в соответствии с порядком своего появления справа налево. Одной и той же опции разрешено появляться несколько раз. Используйте **getopt(3C)** для определения имеющихся опций. *Сначала пусть ваша программа обрабатывает только некоторые опции*. Затем добавьте еще, до тех пор, пока все требуемые опции не будут обрабатываться. Вы можете скопировать воспользоваться программой **getopt_ex.c** и изменить ее.

- ☑ ~~**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(3C)** на странице руководства **strtol(3C)**~~
- ☑ ~~**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.~~

Проверьте вашу программу на различных списках аргументов, в том числе:

- Нет аргументов
- Недопустимую опцию.
- Опции, разделенные знаком минус.
- Неудачное значение для **U.**

Notes

`getopt(3C)` - ***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`

Опции: Должны начинаться со знака минус. Каждая опция представляет собой один символ. Каждая опция может иметь собственный разделительный минус или несколько опций могут совместно использовать один знак минус. Некоторые опции требуют аргументов. Аргумент может следовать непосредственно за символом опции или быть отделен от нее пробелом. Смотрите intro(1) для сверки с общепринятым синтаксисом командной строки.

Другие аргументы: Следуют за опциями и не обязательно должны начинаться со знака минус.

Пример:

```
ls -lt /tmp; pr -n file1 file2;
```

`getopt(3C)` обычно исполняется в начале программы. Она вызывается в цикле для последовательной обработки опций программы.

У `getopt(3C)` три аргумента:
- Целый argc. Обычно первый аргумент main().
- Указатель на вектор символов argv. Обычно второй аргумент main().
- Указатель на символ (строка параметров). Это строка допустимых опций. Если у опции есть аргументы, то за соответствующим символом строки должно стоять двоеточие

`getopt(3C)` возвращает одно из следующих целых значений:

- буква верной опции
- -1 при обработке первого аргумента не опции

`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.
- **optind** целое. Когда `getopt(3C)` возвращает -1, argv[optind] указывает на первый аргумент не-опцию.
  The variable optind is the index of the next element to be processed in argv.
  The system initializes this value to 1.
  The caller can reset it to 1 to restart scanning of the same argv, or when scanning a new argument vector.
- **opterr** целое. Когда `getopt(3C)` обрабатывает недопустимые опции, сообщение об ошибке выводится на стандартный вывод диагностики. Печать может быть подавлена установкой opterr в ноль.
  If getopt() does not recognize an option character, it prints an error message to stderr, stores the character in optopt, and returns '?'.
  The calling program may prevent the error message by setting opterr to 0.
- **optopt** целое. Когда `getopt(3C)` возвращает '?', optopt содержит значение недопустимой опции.

Использование getopt(3C) в программе

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 main(int argc, char *argv[])
5 {

// Создание строки допустимых опций.
// Опции, за которыми следует двоеточие ":", требуют
// соответствующих параметров

 /* OPTSTRING is a string containing the legitimate option characters.
 If an option character is seen that is not listed in OPTSTRING,
 return '?' after printing an error message.  If you set `opterr' to
 zero, the error message is suppressed but we still return '?'.
 If a char in OPTSTRING is followed by a colon, that means it wants an arg,
 so the following text in the same ARGV-element, or the text of the following
 ARGV-element, is returned in `optarg'.  Two colons mean an option that
 wants an optional arg; if there is text in the current ARGV-element,
 it is returned in `optarg', otherwise `optarg' is set to zero*/

6 char options[ ] = "f:dg:"; /* valid options */

// Объявление флагов необязательных опций
7 int c, invalid = 0, dflg = 0, fflg = 0, gflg = 0;
```

```
8 char *f_ptr, *g_ptr;

9 // Печать числа входных параметров.
10 printf("argc equals %d\n", argc);

//Вход в цикл вызова getopt(3C) для просмотра командной строки,
// возвращается одна опция за один проход цикла
11 while ((c = getopt(argc, argv, options)) != EOF) {

// Вход в выбор switch для обработки опций.
// Обычно флаг устанавливается для указания
// присутствия конкретной опции. Флаг используется в программе позже.
12 switch (c) {
13 case 'd':
14 dflg++;
15 break;
16 case 'f':
17 fflg++;
// Если опции нужен параметр, тогда переменная optarg
// будет содержать адрес этого параметра.0
// Если этот адрес будет использоваться позже,
// то он должен быть сохранен в символьном указателе.
18 f_ptr = optarg;
19 break;
20 case 'g':
21 gflg++;
22 g_ptr = optarg;
23 break;
// Если обнаружена недопустимая опция, getopt(3C) вернет '?'
// и выдаст сообщение об ошибке на стандартный вывод диагностики.
// Выдача сообщения может быть выключена установкой opterr в ноль.
// optopt содержит значение недопустимой опции.
24 case '?':
25 printf("invalid option is %c\n", optopt);
26 invalid++;
27 }
28 }
29 printf("dflg equals %d\n", dflg);
30 if(fflg)
31 printf("f_ptr points to %s\n", f_ptr);
32 if(gflg)
33 printf("g_ptr points to %s\n", g_ptr);
34 printf("invalid equals %d\n", invalid);
35 printf("optind equals %d\n", optind);
// Когда getopt(3) возвращает -1, тогда argv[optind]
// указывает на первый аргумент командной строки, отличный от опции.
36 if(optind < argc)
37 printf("next parameter = %s\n", argv[optind]);
38 }
```

What is a env var?

Code:

```
#define _GNU_SOURCE
#include <stdlib.h>
#include <stdio.h>
#include <ulimit.h>
#include <sys/types.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

extern char **environ; // system variable

void showHelp();

int main(int argc, char *argv[]){
  showHelp();
  int currentArg;
  char options[] = "ispuU:cC:dvV:";
  while ((currentArg = getopt(argc, argv, options)) != EOF)
    switch (currentArg){
      case 'i': {
        printf("Real User ID is %d\n", getuid());
        printf("Effective User ID is %d\n", geteuid());
        printf("Real Group ID is %d\n", getuid());
        printf("Effective Group ID is %d\n", getegid());
        break;
      }
```

```
      case 's': {
        // pid == 0 => ID of calling proccess, pgit == 0 => "pid"-proccess is going to lead the group
        // int setpgid(pid_t pid, pid_t pgid);

        if (setpgid(0, 0) != 0){
          perror("failed to set new proccess ID.\n");
        }
        break;
      }

      case 'p':{
        printf("Proccess ID is %d\n", getpid());
        printf("Parent ID is %d\n", getppid());
        // pasing zero means it shall return the process group ID of the calling process
        printf("Group ID is %d\n", getpgid(0));
        break;
      }

      case 'u': {
                                struct rlimit limit;
                                getrlimit(RLIMIT_FSIZE, &limit);
                                printf("Soft file size limit of this process is %ld bytes\n", limit.rlim_cur);
                                printf("Hard file sizr limit of this process os %ld bytes\n", limit.rlim_max);
                                break;
                        }

                        case 'U': {
                                struct rlimit limit;
                                getrlimit(RLIMIT_FSIZE, &limit);
                                limit.rlim_cur = atol(optarg);
                                if (setrlimit(RLIMIT_FSIZE, &limit) != 0){
                                        perror("Any process may decrease its own limit, but only a process with appropriate privileges
                                }
                                break;
                        }

      case 'c': {
        struct rlimit coreFileLimit;
        getrlimit(RLIMIT_CORE, &coreFileLimit);
        printf("Soft core file limit is %lu bytes\n", coreFileLimit.rlim_cur);
        printf("Hard core file limit is %lu bytes\n", coreFileLimit.rlim_max);
        break;
      }

      case 'C': {
        struct rlimit coreFileLimit;
        coreFileLimit.rlim_cur = atol(optarg);
        coreFileLimit.rlim_max = atol(optarg);
        if (setrlimit(RLIMIT_CORE, &coreFileLimit) != 0){
          perror("Any process may decrease its own limit, but only a process with appropriate privileges may increase the limit.\n");
        }
        break;
      }

      case 'd': {
        char* buffer = getcwd(buffer, 200);
        printf("current directory is: %s\n", buffer);
        free(buffer);
        break;
      }

      case 'v': {
        char **p;
        for (p = environ; *p; p++)
          printf ("%s\n", *p);
        break;
      }

      case 'V': {
        if (putenv(optarg) != 0){
          perror("Failed to set new evironment variable\n");
        }
        break;
      }
    }

  return 0;
}

void showHelp(){
  printf("-i    - real and effective user and group ID`s\n"
    "-s    - make current process leader of group\n"
    "-p    - process, parent-process and group\n"
    "-u    - get current ulimit value\n"
    "-Unew_ulimit  - set ulimit to new_ulimit\n"
    "-c    - core file size\n"
    "-Csize    - set core file size to size\n"
```

```
    "-d    - print current directory\n"
    "-v    - show environment variables\n"
    "-Vname=value  - change environment variable or create new if its not exist\n\n");
}
```

▼ getuid(2)

NAME top

```
getuid, geteuid - get user identity
```

SYNOPSIS top

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

uid_t getuid(void);
uid_t geteuid(void);
```

DESCRIPTION top

```
getuid() returns the real user ID of the calling process.
geteuid() returns the effective user ID of the calling process.
```

The distinction between a **real** and an **effective user id** is made because you may have the need to temporarily take another user's identity (most of the time, that would be root, but it could be any user). If you only had one user id, then there would be no way of changing back to your original user id afterwards (other than taking your word for granted, and in case you are root, using root's privileges to change to any user).

So, the real user id is who you really are (the one who owns the process), and the effective user id is what the operating system looks at to make a decision whether or not you are allowed to do something (most of the time, there are some exceptions).

Реальный идентификатор пользователя (или группы) сообщает, кто создал процесс, а эффективный идентификатор пользователя (или группы) сообщает от чьего лица выполняется процесс, если эта информация изменяется. Поскольку решения по ограничению доступа в большей степени зависят от того, от чьего имени выполняется процесс, чем от того, кем он был создан, ядро проверяет эффективные идентификаторы пользователей (или групп) чаще, чем реальные

При входе в систему идентификаторы пользователя и группы устанавливается 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.

Изменить реальный и эффективный идентификаторы пользователя можно с помощью функции `setuid`. Аналогично с помощью функции `setgid` можно изменить реальный и эффективный идентификаторы группы.

```
#include <unistd.h>

int setuid(uid_t uid);

int setgid(gid_t gid);
```

Обе возвращают 0 в случае успеха, −1 — в случае ошибки

Существуют определенные правила, согласно которым изменяются идентификаторы. Рассмотрим их на примере идентификатора пользователя. (Все перечисленное ниже в равной степени относится к идентификатору группы.)

1. Если процесс обладает привилегиями суперпользователя, функция `setuid` устанавливает реальный, эффективный и сохраненный идентификаторы пользователя в соответствии с аргументом *uid*.
2. Если процесс не обладает привилегиями суперпользователя, но аргумент *uid* совпадает с реальным или сохраненным идентификатором пользователя, функция `setuid` изменяет только эффективный идентификатор. Реальный и сохраненный идентификаторы не меняются.
3. Если ни одно из этих условий не соблюдено, `setuid` возвращает значение −1 и записывает в переменную `errno` код ошибки `EPERM`.

Здесь предполагается, что конфигурационный параметр `_POSIX_SAVED_IDS` имеет значение `true`. Если эта функциональная возможность не предоставляется вашей системой, исключите из вышеприведенных правил упоминание о сохраненном идентификаторе.

Сохраненные идентификаторы стали обязательными для реализации в версии POSIX.1 от 2001 года. В более ранних версиях POSIX эта функциональная особенность относилась к разряду необязательных. Чтобы узнать, поддерживается ли она системой, приложение может проверить константу `_POSIX_SAVED_IDS` во время компиляции или вызвать функцию `sysconf` с аргументом `_SC_SAVED_IDS` во время выполнения.

Можно сформулировать несколько правил относительно трех идентификаторов пользователя.

1. Изменить реальный идентификатор пользователя может только процесс, обладающий привилегиями суперпользователя. Как правило, реальный идентификатор пользователя устанавливается программой `login`(1) при входе в систему и никогда не изменяется. Поскольку `login` является процессом, обладающим привилегиями суперпользователя, с помощью функции `setuid` он устанавливает все три идентификатора пользователя.
2. Эффективный идентификатор пользователя устанавливается функцией `exec`, только когда файл программы имеет установленный бит set-user-ID. Если этот бит не установлен, функция `exec` не изменяет эффективный идентификатор пользователя. В любой момент времени можно вызвать функцию `setuid`, чтобы установить эффективный идентификатор равным реальному или сохраненному идентификатору. Но, как правило, нельзя установить эффективный идентификатор пользователя в произвольное значение.
3. Функция `exec` копирует эффективный идентификатор пользователя в сохраненный. Если файл программы имеет установленный бит set-user-ID, эта копия сохраняется после того, как функция `exec` установит эффективный идентификатор равным идентификатору владельца файла.

В табл. 8.7 обобщаются возможные варианты изменения этих трех идентификаторов.

Таблица 8.7. Варианты изменения идентификаторов пользователя

Идентификатор	exec		setuid(uid)	
	Бит set-user-ID выключен	Бит set-user-ID включен	Суперпользователь	Непривилегированный пользователь
Реальный	Не изменяется	Не изменяется	Устанавливается в соответствии с *uid*	Не изменяется
Эффективный	Не изменяется	Устанавливается в соответствии с идентификатором владельца файла программы	Устанавливается в соответствии с *uid*	
Сохраненный	Копия эффективного идентификатора	Копия эффективного идентификатора	Устанавливается в соответствии с *uid*	Не изменяется

Обратите внимание, что с помощью функций `getuid` и `geteuid`, описанных в разделе 8.2, можно получить только текущие значения реального и эффективного идентификаторов пользователя. У нас нет возможности получить текущее значение сохраненного идентификатора.

ERRORS top

```
These functions are always successful.
```

▼ getopt(3C)

```
#include <unistd.h>
int getopt(int argc, char * const argv[],const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

```
The getopt() function parses the command-line arguments.  Its
       arguments argc and argv are the argument count and array as
       passed to the main() function on program invocation.  An element
       of argv that starts with '-' (and is not exactly "-" or "--") is
       an option element.  The characters of this element (aside from
       the initial '-') are option characters.  If getopt() is called
       repeatedly, it returns successively each of the option characters
       from each of the option elements.

       The variable optind is the index of the next element to be
       processed in argv.  The system initializes this value to 1.  The
       caller can reset it to 1 to restart scanning of the same argv, or
       when scanning a new argument vector.

       If getopt() finds another option character, it returns that
       character, updating the external variable optind and a static
       variable nextchar so that the next call to getopt() can resume
       the scan with the following option character or argv-element.

       If there are no more option characters, getopt() returns -1.
       Then optind is the index in argv of the first argv-element that
       is not an option.

       optstring is a string containing the legitimate option
       characters.  If such a character is followed by a colon, the
       option requires an argument, so getopt() places a pointer to the
       following text in the same argv-element, or the text of the
       following argv-element, in optarg.  Two colons mean an option
       takes an optional arg; if there is text in the current argv-
       element (i.e., in the same word as the option name itself, for
       example, "-oarg"), then it is returned in optarg, otherwise
       optarg is set to zero.  This is a GNU extension.  If optstring
       contains W followed by a semicolon, then -W foo is treated as the
       long option --foo.  (The -W option is reserved by POSIX.2 for
       implementation extensions.)  This behavior is a GNU extension,
       not available with libraries before glibc 2.

       By default, getopt() permutes the contents of argv as it scans,
       so that eventually all the nonoptions are at the end.  Two other
       scanning modes are also implemented.  If the first character of
       optstring is '+' or the environment variable POSIXLY_CORRECT is
       set, then option processing stops as soon as a nonoption argument
       is encountered.  If the first character of optstring is '-', then
       each nonoption argv-element is handled as if it were the argument
       of an option with character code 1.  (This is used by programs
       that were written to expect options and other argv-elements in
       any order and that care about the ordering of the two.)  The
       special argument "--" forces an end of option-scanning regardless
       of the scanning mode.

       While processing the option list, getopt() can detect two kinds
       of errors: (1) an option character that was not specified in
       optstring and (2) a missing option argument (i.e., an option at
       the end of the command line without an expected argument).  Such
       errors are handled and reported as follows:

       *  By default, getopt() prints an error message on standard
          error, places the erroneous option character in optopt, and
          returns '?' as the function result.

       *  If the caller has set the global variable opterr to zero, then
          getopt() does not print an error message.  The caller can
          determine that there was an error by testing whether the
          function return value is '?'.  (By default, opterr has a
          nonzero value.)

       *  If the first character (following any optional '+' or '-'
          described above) of optstring is a colon (':'), then getopt()
          likewise does not print an error message.  In addition, it
          returns ':' instead of '?' to indicate a missing option
          argument.  This allows the caller to distinguish the two
          different types of errors.
```

▼ setpgid(2)

```
int setpgid(pid_t pid, pid_t pgid);
```

```
setpgid() sets the PGID of the process specified by pid to pgid.
       If pid is zero, then the process ID of the calling process is
       used.  If pgid is zero, then the PGID of the process specified by
       pid is made the same as its process ID.  If setpgid() is used to
       move a process from one process group to another (as is done by
       some shells when creating pipelines), both process groups must be
       part of the same session (see setsid(2) and credentials(7)).  In
       this case, the pgid specifies an existing process group to be
       joined and the session ID of that group must match the session ID
       of the joining process.
```

Группа процессов - это совокупность процессов с одним и тем же идентификатором группы процессов. Управляющий терминал считает одну из групп процессов в сессии группой основных процессов. Все процессы в основной группе будут получать сигналы, относящиеся к терминалу, такие как `SIGINT` и `SIGQUIT`. Новый идентификатор группы процессов может быть создан вызовом `setpgid(2)`. Группы процессов, отличные от основной группы той же сессии, считаются группами фоновых процессов. `ksh` использует группы процессов для управления заданиями. Фоновые процессы не получают сигналов, генерируемых терминалом. Системный вызов `setpgid(2)` устанавливает идентификатор группы процессов следующим образом:

pid== pgid создается группа процессов с идентификатором, равным `pid`; вызвавший процесс становится лидером этой группы

pid != pgid процесс `pid` становится членом группы процессов `pgid`, если она существует и принадлежит к этой сессии.

Если `pid` равен 0, будет использован идентификатор вызывающего процесса.
Если `pgid` равен нулю, процесс с идентификатором `pid` станет лидером группы процессов. `pid` должен задавать процесс, принадлежащий к той же сессии, что и вызывающий.

В `ksh` и `bash` для каждой исполняемой команды создается новая группа процессов.
В `sh` все процессы принадлежат к одной группе, если только сам процесс не исполнит `setsid(2)` или `setpgid(2)`.

> 📢 **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.

что за ksh sh bash?

▼ setrlimit getrlimit

7.11. Функции getrlimit и setrlimit

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 */
};
```

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.

▼ getpid(2) & getppid(2)

```
getpid() returns the process ID (PID) of the calling process.
       (This is often used by routines that generate unique temporary
       filenames.)

getppid() returns the process ID of the parent of the calling
       process.  This will be either the ID of the process that created
       this process using fork(), or, if that process has already
       terminated, the ID of the process to which this process has been
       reparented (either init(1) or a "subreaper" process defined via
       the prctl(2) PR_SET_CHILD_SUBREAPER operation).
```

- 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.

▼ get_current_dir_name(3)

```
#include <unistd.h>
   char *get_current_dir_name(void);

get_current_dir_name() will malloc(3) an array big enough to hold
       the absolute pathname of the current working directory.
If the environment variable PWD is set, and its value
is correct, then that value will be returned.
The caller should free(3) the returned buffer.
```

▼ core file

Дамп памяти (англ. memory dump; в Unix — core dump) — содержимое рабочей памяти одного процесса, ядра или всей операционной системы. Также может включать дополнительную информацию о состоянии программы или системы, например значения регистров процессора и содержимое стека. Многие операционные системы позволяют сохранять дамп памяти для отладки программы. Как правило, дамп памяти процесса сохраняется автоматически, когда процесс завершается из-за критической ошибки (например, из-за ошибки сегментации). Дамп также можно сохранить вручную через отладчик или любую другую специальную программу.

Английский термин core dump буквально переводится как «выгрузка содержимого ядра»: на ранних компьютерах дамп означал принтерную распечатку содержимого памяти на магнитных сердечниках (англ. magnetic core memory). образ оперативной памяти программы, сохраненный в файле на диске для последующего анализа

В современных Unix-подобных операционных системах дамп памяти сохраняется в виде файла, который обычно называется core или core.<номер процесса>; его формат такой же, как формат исполняемых файлов этой ОС (ELF в Linux и современных Unix, a.out в традиционных Unix-системах, Mach-O в Mac OS X). Для анализа core-файла используется отладчик (например gdb) или инструмент objdump.

Core file and crash dumps are generated when a process or application terminates abnormally. You must configure your system to allow Directory Server to generate a core file if the server crashes. The core file contains a snapshot of the Directory Server process at the time of the crash, and can be indispensable in determining what led to the crash. Core files are written to the same directory as the errors logs, by default, instance-path/logs/. Core files can be quite large, as they include the entry cache.

Questions

Порожденные процессы наследуют от родительского процесса идентификаторы группы процессов, сессии и управляющий терминал.
Группы процессов и сессии важны для управлениями заданиями и обработки сигналов. В частности, у терминала в каждый момент времени есть «основная» группа процессов.
Только процессы этой группы могут читать данные с терминала.

https://docs.oracle.com/cd/E23824_01/html/821-1451/userconcept-23295.html

https://github.com/illumos/illumos-gate/blob/master/usr/src/cmd/sh/ulimit.c

Reading list

- [ ] core files
- [ ] ulimit; file size limits soft and hard