Програми на C, използващи системни извиквания

Операционни системи, ФМИ, 2022/2023

Увод

В този раздел ще пишем на C.

Неща, които трябва да знаете, преди да започнем:

- Синтаксис на C
- Как компилираме C код
- Как пускаме програми на C

Езикът C и системни извиквания

Операционни системи, ФМИ, 2022/2023

Системни извиквания

- Дотук знаем, че операционната система имплементира комуникация между процеси и връзка на процесите с външния свят
- Връзката между един процес и (ядрото на) операционната система се извършва чрез операции, наречени *системни извиквания* (system calls или syscalls)
- В този раздел ще се научим да пишем програми, които директно използват системните извиквания на операционната система
  - Това се нарича *“системно програмиране”*

Как работят системните извиквания

- От гледна точка на програмиста, системните извиквания са “*просто*” библиотечни функции, които може да извика
- Специалното на тези функции е, че вместо да изпълняват код като част от програмата, *казват* на ядрото да изпълни съответната операция и чакат резултат
- Програмата заспива докато ядрото не стане готово с изпълнението на системното извикване, и когато се събуди, получава резултат
- Тъй като тези функции “обвиват” системните извиквания, ще ги наричаме *syscall wrappers*

Как работят системните извиквания

- Най-често използваната C библиотека, имплементираща syscall wrappers, е **glibc**
  - glibc е “стандартна библиотека” за C, която освен syscall wrappers имплементира и всички функции от C стандарта
  - Има и други такива библиотеки, напр. musl
- Съответните библиотеки съдържат хедъри със стандартизирани имена (`fcntl.h`, `unistd.h`, ...), в които са дефинирани syscall wrapper-ите

Какво реално прави един syscall wrapper?

- Копира аргументите на системното извикване и неговия номер в специфични процесорни регистри
- Изпълнява процесорна инструкция, която предизвиква хардуерно прекъсване
  - При стартиране на операционната система, тя е конфигурирала процесора при такова прекъсване да скочи на специфично място в кода на ядрото, където ще се прочетат въпросните аргументи и номер на системно извикване, и то ще се обработи
  - След това ядрото записва резултата от системното извикване в специфичен процесорен регистър и скача обратно в кода на syscall wrapper-а
- Вади резултата от въпросния процесорен регистър и го връща

Какво реално прави един syscall wrapper?

- Когато имам време, на този слайд ще сложа картинка

Защо C?

- Езикът, на който е написан Linux (и други UNIX-и) е C
- По тази причина, системните извиквания използват подреждане и формат на данните, съобразени с ABI[1] на C
- Затова повечето библиотеки, имплементиращи syscall wrappers, са написани на C
  - Нищо не ни пречи да използваме системни извиквания директно и от друг език - пример за език, който има собствена имплементация на syscall wrappers е Go
  - На практика почти всички останали езици извикват C код през FFI, който от своя страна вика syscall wrappers от библиотека като `glibc`, вместо да ги имплементират сами

[1] ABI: Application Binary Interface

Пример за системни извиквания на C

Системните извиквания `getuid` и `geteuid` връщат реалното и ефективното UID на процеса, изпълняващ програмата.

```c
#include <unistd.h>     // getuid, geteuid
#include <sys/types.h>  // uid_t
#include <stdio.h>      // printf

int main(void)
{
    uid_t me = getuid();
    uid_t pretending = geteuid();
    printf("uid: %d euid: %d\n", me, pretending);
    return 0;           // exit status 0
}
```

man syscall

В този раздел ще са ни полезни следните секции от man страници:

- Секция 2: системни извиквания
  - `getuid(2)`
  - `geteuid(2)`
- Секция 3: библиотечни функции в C
  - `printf(3)`

Exit status

Exit status-ът на една програма е стойността, върната от функцията `main()`:

```
int main(void)
{
    return 42;              // exit status 42
}
```

Exit status

Системното извикване `_exit(2)` прекратява изпълнението на процеса, независимо от текущата функция:

```c
#include <unistd.h>     // _exit

void foo(void) {
    _exit(42);      // exit status 42
}

int main(void)
{
    foo();
}
```

Exit status

Библиотечната функция `exit(3)` вътрешно вика `_exit(2)`, но финализира и някои други неща преди това.

Използвайте нея, когато искате да прекратите програмата.

```
#include <stdlib.h>     // exit

void foo(void) {
    exit(42);       // exit status 42
}

int main(void)
{
    foo();
}
```

Обработване на грешки при системни извиквания

Операционни системи, ФМИ, 2022/2023

Резултат от системни извиквания

- По конвенция, повечето системни извиквания връщат резултат от числов тип, който е отрицателно число, ако извикването е било неуспешно
- За пример ще използваме `open(2)` - системно извикване, отварящо файл

Резултат от системни извиквания

```
int open(const char *pathname, int flags)
```

- първият аргумент е път до файл
- вторият аргумент е множество от опции, задаващи режима на отваряне на файл
- резултатът при успех е положително число - номер на *файлов дескриптор* - след малко ще говорим за него
- засега важното е, че резултатът от `open()` е `-1`, ако отварянето на файл е било неуспешно

Грешки при системни извиквания

```c
#include <fcntl.h>  // open
#include <stdio.h>  // printf
#include <stdlib.h> // exit

int main() {
    int result = open("/tmp/some_file", O_RDONLY);
    if (result < 0) {
        printf("opening /tmp/some_file failed\n");
        exit(1);
    }

    printf("opened /tmp/some_file successfully\n");
}
```

errno

- Когато някое системно извикване е неуспешно, ни се иска начин да разберем какво не е било наред
- Затова, при неуспех системните извиквания записват число (код на грешка) в **глобалната променлива** `errno`
  - Вижте `errno(3)` за повече информация

errno

Можем да използваме `errno`, за да разберем каква е била грешката:

```c
#include <fcntl.h>  // open
#include <stdio.h>  // printf
#include <errno.h>  // errno
#include <stdlib.h> // exit

int main() {
    int result = open("/tmp/some_file", O_RDONLY);
    if (result < 0) {
        switch (errno) {
            case 2:  printf("no such file\n"); break;
            case 13: printf("permission denied\n"); break;
            // ...
        }
        exit(1);
    }

    printf("opened /tmp/some_file successfully\n");
}
```

errno

По-добре е да ползваме константите, дефинирани в `errno.h`, вместо чисти номера на грешки:

```c
#include <fcntl.h>  // open
#include <stdio.h>  // printf
#include <errno.h>  // errno
#include <stdlib.h> // exit

int main() {
    int result = open("/tmp/some_file", O_RDONLY);
    if (result < 0) {
        switch (errno) {
            case ENOENT: printf("no such file\n"); break;
            case EACCES: printf("permission denied\n"); break;
            // ...
        }
        exit(1);
    }

    printf("opened /tmp/some_file successfully\n");
}
```

err.h

Най-добрият вариант е да използвате функцията `err()`, която изписва форматирано съобщение за грешка (вътрешно гледа променливата errno).

Първият аргумент на `err()` е exit status, с който да прекрати програмата.

```c
#include <fcntl.h>  // open
#include <stdio.h>  // printf
#include <err.h>    // err

int main() {
    int result = open("/tmp/some_file", O_RDONLY);
    if (result < 0) {
        err(1, "could not open file");
    }

    printf("opened /tmp/some_file successfully\n");
}
```

err.h

```c
#include <fcntl.h>  // open
#include <stdio.h>  // printf
#include <err.h>    // err

int main() {
    int result = open("/tmp/some_file", O_RDONLY);
    if (result < 0) {
        err(1, "could not open file");
    }

    printf("opened /tmp/some_file successfully\n");
}
```

Например при грешка ENOENT, тази програма ще изведе съобщение, изглеждащо така:

```
could not open file: No such file or directory
```

Вторият и следващите аргументи на `err()` задават форматин низ (както при `printf()`):

```c
#include <fcntl.h>  // open
#include <stdio.h>  // printf
#include <err.h>    // err

int main() {
    const char filename[] = "/tmp/some_file";
    int result = open(filename, O_RDONLY);
    if (result < 0) {
        err(1, "could not open file %s", filename);
    }

    printf("opened %s successfully\n", filename);
}
```

Например при грешка `ENOENT`, тази програма ще изведе съобщение, изглеждащо така:

```
could not open file /tmp/some_file: No such file or directory
```

err.h

Всъщност, `err.h` задава 4 полезни функции:

- `void err(int eval, const char* fmt, ...)`
  - изписва съобщението, което сме подали
  - изписва грешката от `errno`
  - прекратява програмата с подадения статус
- `void errx(int eval, const char* fmt, ...)`
  - изписва съобщението, което сме подали
  - **не** изписва грешката от `errno`
  - прекратява програмата с подадения статус
- `void warn(const char* fmt, ...)`
  - изписва съобщението, което сме подали
  - изписва грешката от `errno`
  - **не** прекратява програмата
- `void warnx(const char* fmt, ...)`
  - изписва съобщението, което сме подали
  - **не** изписва грешката от `errno`
  - **не** прекратява програмата

Ще срещнете и четирите вида ситуации, в които е подходяща съответната функция. Ползвайте ги!

Файлови дескриптори

Операционни системи, ФМИ, 2022/2023

Файлови дескриптори

- Споменахме, че системното извикване `open()` връща число, което наричаме *номер на файлов дескриптор*
- При отваряне на файл, ядрото създава системна структура, наречена *файлов дескриптор*, която съдържа:
  - Указател към самия файл
  - Текуща позиция (индекс на байт) във файла
  - И други
- За всеки процес ядрото алокира масив от файлови дескриптори: номерът на файлов дескриптор, върнат от `open()`, е индекс в този масив.

Опции на open()

- `open()` може да приема 2 или 3 аргумента:

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- Аргументите на `open()` са:
  - Път до файл
  - Множество от опции
  - Права за достъп (mode) - в осмична бройна система

Опции на open()

Вторият аргумент на `open()` е *битова маска* от опции.

- Комбинираме опциите с “побитово ИЛИ”
- Някои опции, които ще ни трябват, са:
  - `O_WRONLY`, `O_RDONLY` - отваряне за писане или за четене
  - `O_RDWR` - отваряне за четене и писане едновременно
  - `O_CREAT` - ако файлът не съществува, го създава преди да го отвори
  - `O_TRUNC` - ако файлът съществува, зачиства съдържанието му преди да го отвори
  - `O_APPEND` - ако файлът съществува, началната позиция е в края му вместо в началото

Опции на open()

Третият аргумент на `open()` задава права за достъп на файла, ако го създаваме сега

- Има смисъл само с `O_CREAT`
- Единият вариант е да зададем правата директно като число

```
int fd = open(
    "/tmp/some_file",
    O_WRONLY|O_CREAT|O_TRUNC,
    0644
);
```

- Другият вариант е да ползваме побитови константи, дефинирани в стандартната библиотека

```
int fd = open(
    "/tmp/some_file",
    O_WRONLY|O_CREAT|O_TRUNC,
    S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH
);
```

- За повече информация, вижте `open(2)`

Опции на open(): примери

- Отваряне на файл за четене

```
int fd = open("/tmp/some_file", O_RDONLY);
```

- Отваряне на файл за писане, създавайки го с права 644 ако не съществува и зачиствайки го, ако съществува

```
int fd = open(
    "/tmp/some_file",
    O_WRONLY|O_CREAT|O_TRUNC,
    0644
);
```

- Отваряне на файл за писане, създавайки го с права 644 ако не съществува и застaвайки накрая му, ако съществува

```
int fd = open(
    "/tmp/some_file",
    O_WRONLY|O_CREAT|O_APPEND,
    0644
);
```

Затваряне на отворен файл

- Когато един процес умре, всички негови файлови дескриптори се затварят автоматично
- Можем ръчно да затворим файлов дескриптор, използвайки системното извикване `close(2)`

```c
#include <fcntl.h>  // open, close
#include <err.h>    // err

int main() {
    const char filename[] = "/tmp/some_file";
    int fd = open(filename, O_RDONLY);
    if (fd < 0) {
        err(1, "could not open file %s", filename);
    }

    // ...

    if (close(fd) < 0) {
        err(1, "could not close file %s", filename);
    }
}
```

Затваряне на отворен файл

- Добре е отворените файлове да се затварят веднага, щом сме приключили да работим с тях

Четене от файл

- Можем да четем от отворен файл със системното извикване `read(2)`:

```
char buf[20];
int num_bytes = read(fd, buf, 20);
if (num_bytes < 0) {
    err(1, "could not read data");
}
```

- Аргументите на `read()` са:
  - Номер на файлов дескриптор, от който четем
  - Указател към памет, в която искаме да се запишат прочетените данни
  - Максимална дължина на четене
- Резултатът от `read()` е броят реално прочетени байтове

Четене от файл

- При успешно изпълнение на `read()`, текущата позиция на файловия дескриптор се придвижва напред с броя успешно прочетени байтове
- Това означава, че всяко следващо викане на `read()` чете нови данни от файла
- Ако `read()` прочете 0 байта, това означава, че сме стигнали края на файла

Четене от файл

```
int fd = open("/tmp/some_file", O_RDONLY);
if (fd < 0) {
    err(1, "could not open file");
}

char buf[20];
int num_bytes = read(fd, buf, 20);
while (num_bytes > 0) {
    // < do something with the data in buf >
    num_bytes = read(fd, buf, 20);
}
if (num_bytes < 0) {
    err(1, "could not read from file");
}

if (close(fd) < 0) {
    err(1, "could not close file");
}
```

Четене от файл

```
int fd = open("/tmp/some_file", O_RDONLY);
if (fd < 0) {
    err(1, "could not open file");
}

char buf[4096];
int num_bytes;
while ((num_bytes = read(fd, buf, sizeof(buf))) > 0) {
    // < do something with the data in buf >
}
if (num_bytes < 0) {
    err(1, "could not read from file");
}

if (close(fd) < 0) {
    err(1, "could not close file");
}
```

Четене от файл

```
int fd = open("/tmp/some_file", O_RDONLY);
if (fd < 0) {
    err(1, "could not open file");
}

char c;
int num_bytes;
while ((num_bytes = read(fd, &c, 1)) > 0) {
    // < do something with the character in c >
}
if (num_bytes < 0) {
    err(1, "could not read from file");
}

if (close(fd) < 0) {
    err(1, "could not close file");
}
```

Писане във файл

Писането във файл е аналогично на четенето: чрез системното извикване `write(2)`:

```
char buf[] = "Hello world!\n";
int num_bytes = write(fd, buf, 13);
if (num_bytes < 0) {
    err(1, "could not write data");
}
if (num_bytes != 13) {
    errx(1, "could not write data all at once");
}
```

- Аргументите на `write()` са:
  - Номер на файлов дескриптор, в който пишем
  - Указател към памет, от която да се прочетат данни
  - Брой байтове (размер на данните), които искаме да запишем
- Резултатът от `write()` е броят реално записани байтове

Записване на текстови низове от паметта в текстов файл

- При записване на текстови низове, трябва да внимаваме да не запишем терминращия нулев символ във файла
  - Функцията `strlen(3)` е полезна: връща размера на низа, без да включва терминиращата нула

```
char buf[] = "Hello world!\n";
int num_bytes = write(fd, buf, strlen(buf));
if (num_bytes < 0) {
    err(1, "could not write data");
}
if (num_bytes != 13) {
    errx(1, "could not write data all at once");
}
```

Файлови дескриптори на стандартни потоци

- Стандартните потоци `stdin`, `stdout` и `stderr` по подразбиране съществуват при създаване на процес
  - Файлов дескриптор `0` е stdin
  - Файлов дескриптор `1` е stdout
  - Файлов дескриптор `2` е stderr

Файлови дескриптори на стандартни потоци

```
char name_buf[512];
const char prompt[]   = "What's your name? ";
const char hello[]    = "Hello, ";
const char end[]      = "!\n";

int write_result = write(1, prompt, strlen(prompt));
if (write_result < 0) { err(1, "could not write prompt"); }

int name_len = read(0, name_buf, strlen(name_buf));
if (name_len < 0) { err(1, "could not read name"); }

write_result = write(1, hello, strlen(hello));
if (write_result < 0) { err(1, "could not write hello"); }

write_result = write(1, name_buf, strlen(name_buf));
if (write_result < 0) { err(1, "could not write name"); }

write_result = write(1, end, strlen(end));
if (write_result < 0) { err(1, "could not write end"); }
```

Преместване на текущата позиция във файл

- Досега видяхме, че `read()` и `write()` местят текущата позиция *напред*
- Със системното извикване `lseek(2)` можем да преместим текущата позиция на *произволно място* във файла

```
off_t lseek(int fd, off_t offset, int whence);
```

Преместване на текущата позиция във файл

```
off_t lseek(int fd, off_t offset, int whence);
```

- Аргументите на `lseek()` са:
  - Файлов дескриптор
  - Отместване
  - Интерпретация на отместването
- Резултатът от `lseek()` е новата абсолютна позиция
- Възможните интерпретации на отместването (`whence`) са:
  - `SEEK_SET`: абсолютно отместване
  - `SEEK_CUR`: относително отместване спрямо текущата позиция
  - `SEEK_END`: относително отместване спрямо края на файла

Преместване на текущата позиция във файл

- Скачане в началото на файла:

```
int new_pos = lseek(fd, 0, SEEK_SET);
if (new_pos < 0) {
    err(1, "could not go to start of file");
}
```

- Скачане на позиция 42 във файла:

```
int new_pos = lseek(fd, 42, SEEK_SET);
if (new_pos < 0) {
    err(1, "could not go byte 42");
}
```

- Скачане с 5 байта назад:

```
int new_pos = lseek(fd, -5, SEEK_CUR);
if (new_pos < 0) {
    err(1, "could not jump 5 bytes backwards");
}
```

Преместване на текущата позиция във файл

- Скачане 2 байта преди края на файл:

```
int new_pos = lseek(fd, -2, SEEK_END);
if (new_pos < 0) {
    err(1, "could not jump to 2 bytes before end");
}
```

- Скачане 42 байта след края на файл:
  - Това работи само ако можем да пишем във файла
  - Файлът пораства с толкова, колкото е нужно

```
int new_pos = lseek(fd, 42, SEEK_END);
if (new_pos < 0) {
    err(1, "could not jump to 42 bytes after end");
}
```

Четене и писане на двоични данни от паметта във файлове

Операционни системи, ФМИ, 2022/2023

Форматиран и неформатиран вход/изход

- Мислено можем да разделим подходите за вход/изход на две категории:
  - Форматиран вход/изход
  - Неформатиран вход/изход

Неформатиран вход/изход

- Когато говорим за *неформатиран* вход/изход, имаме предвид, че програмата чете и пише *данни* във формат, който не може да се интерпретира като текст
- *Числата* най-често ги представяме по същия начин, както са представени в паметта
- Системните извиквания `read()` и `write()` могат да се използват за неформатиран вход/изход на данни в паметта, без промяна на тяхната структура

Неформатиран вход/изход

```c
void write_number(int fd, uint16_t num) {
    write(fd, &num, sizeof(num));
}

uint16_t read_number(int fd) {
    uint16_t num;
    read(fd, &num, sizeof(num));
    return num;
}
```

Форматиран вход/изход

- Когато говорим за *форматиран* вход/изход, имаме предвид, че програмата чете и пише *текст*, предназначен за четене от хора
- Нищо не пречи текстът да е и машинно четим
  - това го обсъдихме по-надълго в темата “Данни във файлове”
- Числата са *форматирани* като последователности от цифри (текст)

Форматиран изход на числа

```
int n_digits(uint16_t num) {
    if (num == 0) { return 1; }
    int result = 0;
    for (; num != 0; num /= 10) { result++; }
    return result;
}
void num_to_text(uint16_t num, char* buf) {
    buf[n_digits(num)] = '\0';
    for (int i = n_digits(num) - 1; i >= 0; i--) {
        buf[i] = '0' + (num % 10);
        num /= 10;
    }
}
void print_number(int fd, uint16_t num) {
    char num_text[6];
    num_to_text(num, num_text);
    write(fd, num_text, n_digits(num));
}
```

Форматиран изход на числа

- Можем да използваме вградената функция `snprintf(3)` за да форматираме числа като текст:

```
void print_number(int fd, uint16_t num) {
    char num_text[6];
    snprintf(num_text, sizeof(num_text), "%d", num);
    write(fd, num_text, n_digits(num));
}
```

Работим само с файлови дескриптори

- В стандартната библиотека на C има абстракция за работа с файлове, наречена `FILE*`, която обвива системните извиквания за работа с входно-изходни операции във функции от по-високо ниво
- Повечето такива функции са в `<stdio.h>`
- **Ние няма да ползваме тези функции**, а ще работим директно с файловите дескриптори
- Единствените функции от `<stdio.h>`, които ще си позволим, са:
  - `dprintf()`, за форматиран изход (само в случаите, в които искаме да форматираме число)
  - `snprintf()`, за генериране на форматиран низ

Информация за файловете чрез `stat`

Операционни системи, ФМИ, 2022/2023

Информация за файловете чрез `stat`

- Системното извикване `stat(2)` дава достъп до метаданните на файла (командата `stat(1)` използва това системно извикване)
- Първият аргумент е път до файл, а вторият е указател към структура от тип `struct stat`, която е дефинирана в стандартната библиотека.

struct stat - stat(3type)

```
struct stat {
    dev_t      st_dev;     // ID of device containing file
    ino_t      st_ino;     // Inode number
    mode_t     st_mode;    // File type and mode
    nlink_t    st_nlink;   // Number of hard links
    uid_t      st_uid;     // User ID of owner
    gid_t      st_gid;     // Group ID of owner
    dev_t      st_rdev;    // Device ID (if special file)
    off_t      st_size;    // Total size, in bytes
    blksize_t  st_blksize; // Block size for filesystem I/O
    blkcnt_t   st_blocks;  // Number of 512 B blocks allocated

    struct timespec  st_atim;  // Time of last access
    struct timespec  st_mtim;  // Time of last modification
    struct timespec  st_ctim;  // Time of last status change
};
```

stat

```c
struct stat info;
int result = stat("/tmp/foo.txt", &info);
if (result < 0) { err(1, "could not stat /tmp/foo.txt") };

dprintf(1, "owner UID: %d\n", info.st_uid);
dprintf(1, "owner GID: %d\n", info.st_gid);
dprintf(1, "size: %d bytes\n", info.st_size);
```

stat

- Алтернативният вариант `fstat()` приема файлов дескриптор като първи аргумент вместо път
- Може да го ползвате за вече отворени файлове

Изпълняване на програми с exec

Операционни системи, ФМИ, 2022/2023

Изпълняване на програми с exec

- Фамилията от системни извиквания `exec(3)` се използва, за да изпълним външна програма в текущия процес
  - Различни варианти на извикване - `execl()`, `execlp()`, `execvp()`, `execle()`, `execve()`...
- При успешно изпълнение на `exec()`, програмата на текущия процес се **заменя** с дадената

Изпълняване на програми с exec

```c
int main(void) {
    int result = execl(
        "/usr/bin/cat",          // executable
        "cat", "/etc/issue",     // arguments
        (char*)NULL              // sentinel
    );

    if (result < 0) {
        err(1, "could not exec");
    }

    dprintf(1, "you will never read this\n");
}
```

exec*p - търсене в $PATH

Вариантите exec*p използват environment променливата PATH за да търсят изпълнимия файл:

```
int main(void) {
    int result = execlp(
        "cat",                  // executable
        "cat", "/etc/issue",    // arguments
        (char*)NULL             // sentinel
    );

    if (result < 0) {
        err(1, "could not exec");
    }

    dprintf(1, "you will never read this\n");
}
```

execv* - масив от аргументи

Вариантите execv* приемат масив от аргументи:

```
int main(int argc, char* argv[]) {
    if (argc > 9) {
        errx(1, "cannot work with more than 8 arguments");
    }

    char* command_args[10];
    char cat[] = "cat";
    command_args[0] = &cat;

    for (int i = 1; i < argc; i++) {
        command_args[i] = argv[i];
    }
    command_args[argc] = NULL;

    execvp("cat", command_args);
    err(1, "could not exec cat");
}
```

Създаване на процеси

Операционни системи, ФМИ, 2022/2023

Създаване на процеси

- В UNIX света създаването на процеси става чрез системното извикване `fork(2)`.
- При извикване на `fork()`, текущият процес се клонира на *родител* и *дете*
  - Семантично, цялата памет на процеса се **копира**
  - Реално копието се извършва чрез *copy-on-write* (CoW)
- Родителят и детето използват **отделни** региони във физическата памет и не могат да достъпват паметта по между си
- Изпълнението на програмата в процеса-дете продължава от мястото, където е извикан `fork()`
- Отделните процеси работят конкурентно и не се изчакват

Създаване на процеси

- Стойността, върната от `fork()`, е различна при родителя и детето:
  - В детето, `fork()` връща 0
  - В родителя, `fork()` връща число, по-голямо от 0: `pid`-а на детето

Създаване на процеси

```
pid_t pid = fork();
if (pid < 0) {
    err(1, "could not fork");
}

if (pid > 0) {
    dprintf(1, "I am your father\n");
} else {
    dprintf(1, "Noooooooooo!\n");
}
```

Създаване на процеси

```
pid_t pid = fork();
if (pid < 0) {
    err(1, "could not fork");
}

if (pid > 0) {
    dprintf(1, "I am the parent\n");
    dprintf(1, "The child's pid is %d\n", pid);
} else {
    dprintf(1, "I am the child\n");
}

dprintf(1, "I am both\n");
```

pids

- Както видяхме, PID-ът на процеса-дете се връща от `fork()`
- `getpid(2)` и `getppid(2)` връщат PID-а на текущия процес и на неговия родител:

```
pid_t my_pid = getpid();
pid_t parent_pid = getppid();

dprintf(
    1,
    "My pid is %d and my parent's pid is %d\n",
    my_pid, parent_pid
);
```

Изчакване с `wait()`

- Системното извикване `wait(2)` блокира, докато някое дете на текущия процес умре
  - Аргументът му е указател, сочещ към променлива, в която `wait()` ще запише статуса на завършилото дете
  - Стойността, върната от `wait()` е `PID`-а на детето

Изчакване с `wait()`

- Всъщност, статусът, който `wait()` записва в аргумента си, кодира малко повече информация освен exit status-а на процеса-дете
- Например, можем да разберем дали процесът е бил убит или е завършил нормално
  - Макрото `WIFEXITED(status)` проверява дали статусът е такъв на нормално-завършил процес
- Можем и да извлечем истинският exit status на процеса
  - Макрото `WEXITSTATUS(status)` извлича exit status-а
- За по-подробна информация, вижте `wait(2)`

Изчакване с wait()

```
for (int i = 0; i < num_tasks; i++) {
    pid_t child_pid = fork();
    if (child_pid < 0) { err(1, "could not fork"); }
    if (child_pid == 0) {
        do_task(i);
        exit(0);    // The child does its work and exits
    }
}

for (int i = 0; i < num_tasks; i++) {
    int status;
    pid_t child_pid = wait(&status);
    if (child_pid < 0) { err(1, "could not wait for child")
    if (!WIFEXITED(status)) {
        warnx(1, "a task failed: child was killed!");
    } else  if (WEXITSTATUS(status) != 0) {
        warnx(1, "a task failed (exit status != 0)!");
    }
}

dprintf(1, "all tasks completed successfully\n");
```

Изчакване с `wait()`

- Със системното извикване `waitpid(2)` можем да изчакаме завършването на процес със конкретен `PID`
- Има и малко повече възможности от `wait()`
  - Може да провери дали процес е завършил, без да блокира
  - Може да чака за цяла група процеси
- За информация как да го ползвате, вижте man страницата.

Наследяване на средата при `fork()`

- Процесът-дете наследява цялата среда на родителя си
  - Потребител (`EUID`, `UID`)
  - Права
  - Environment променливи
  - Отворени файлови дескриптори

Наследяване на файлови дескриптори

```
int fd = open(
    "/tmp/test.txt",
    O_WRONLY|O_CREAT|O_TRUNC,
    0666
);
if (fd < 0) { err(1, "could not open file"); }

pid_t pid = fork();
if (pid < 0) { err(1, "could not fork"); }

for (int i = 0; i < 1000; i++) {
    if (pid == 0) {
        write(fd, "foo\n", 4);
    } else {
        write(fd, "bar\n", 4);
    }
}

close(fd);
```

Наследяване на файлови дескриптори

- В предния пример получихме файл, в който имаме 1000 реда “foo” и 1000 реда “bar”, в произволен ред
- Двата процеса имат достъп до един и същ файлов дескриптор
- Процесите се състезават, кой от тях да запише своя текст и да премести указателя на файловия дескриптор напред
- В следващата тема ще се възползваме от наследяване на файлови дескриптори, за да правим по-интересни неща

Тръби и водопроводчици

Операционни системи, ФМИ, 2022/2023

pipe

- Системното извикване `pipe(2)` създава *тръба*
  - Тръбата е структура в ядрото, имплементираща FIFO опашка
- Взаимодействаме с тръбата през два файлови дескриптора:
  - `pipe()` приема като аргумент масив от 2 елемента, в който да запише номерата на двата файлови дескриптора
  - Дескриптор за четене (индекс 0)
  - Дескриптор за писане (индекс 1)

pipe

```c
int pfd[2];
if (pipe(pfd) < 0) {
    err(1, "could not create pipe");
}

pid_t pid = fork();
if (pid < 0) { err(1, "could not fork"); }
if (pid == 0) {
    close(pfd[0]);

    write(pfd[1], "foo\n", 4);

    close(pfd[1]);
    exit(0);
} else {
    close(pfd[1]);

    char buf[20];
    read(pfd[0], buf, 20);
    // ... do something with data
}
```

pipe

- Тръбите са удобен метод за комуникация между процеси
- При четене от тръбата, текущият процес *блокира* докато някой друг не запише данни в тръбата
- Когато всички краища за писане се затворят, краищата за четене получават “край на файл” (EOF) и четенето от тях вече не блокира
- Нужно е всеки процес да затваря краищата на тръбата, които не ползва
  - В противен случай може да се получи deadlock
  - Например, може процесът, който пише данни, да е приключил, но процесът, който чете данни, да чака блокиран до безкрай, защото не е затворил своя край за писане

Копиране на файлови дескриптори

- Можем да копираме файлови дескриптори със системните извиквания `dup()` и `dup2()`: вижте `dup(2)`
- `int dup(int oldfd)` - копира подадения файлов дескриптор с номер `oldfd` на първия свободен номер, и връща новия номер
- `int dup2(int oldfd, int newfd)` - копира подадения файлов дескриптор с номер `oldfd` като нов файлов дескриптор с номер `newfd` и връща `newfd`
  - Ако файлов дескриптор с номер `newfd` е съществувал, `dup2()` го затваря преди да направи копието
- Тези системни извиквания са много полезни, ако искаме да имплементираме пренасочване на стандартните потоци

Пренасочване

Пренасочване на изхода на команда към файл:

```
int fd = open(
    "/tmp/test.txt",
    O_WRONLY|O_CREAT|O_TRUNC,
    0666
);
if (fd < 0) { err(1, "could not open file"); }

int result = dup2(fd, 1);   // replace stdout by fd
if (result < 0) { err(1, "could not dup"); }

execlp("ps", "ps", "-e", (char*)NULL);
err(1, "could not exec");
```

Пренасочване: pipe + dup

Пренасочване на текст към `stdin` на команда:

```
int pfd[2];
if (pipe(pfd) < 0) {
    err(1, "could not create pipe");
}

pid_t pid = fork();
if (pid < 0) { err(1, "could not fork"); }
if (pid == 0) {
    close(pfd[0]);

    write(pfd[1], "foo\n", 4);
    close(pfd[1]);
    exit(0);
} else {
    close(pfd[1]);

    dup2(pfd[0], 0); // replace stdin by pfd[0]
    execlp("wc", "wc", "-m", (char*)NULL);
    err(1, "could not exec wc");
}
```

pipe + dup

Пренасочване на изхода на една команда към входа на друга:

```
int pfd[2];
if (pipe(pfd) < 0) {
    err(1, "could not create pipe");
}

pid_t pid = fork();
if (pid < 0) { err(1, "could not fork"); }
if (pid == 0) {
    close(pfd[0]);
    dup2(pfd[1], 1); // replace stdout by pfd[1]
    execlp("ps", "ps", "-e", (char*)NULL);
    err(1, "could not exec ps");
} else {
    close(pfd[1]);
    dup2(pfd[0], 0); // replace stdin by pfd[0]
    execlp("grep", "grep", "firefox", (char*)NULL);
    err(1, "could not exec grep");
}
```

pipe + dup

- Всъщност, shell-ът имплементира операторите >, >>, < и | точно така: чрез `pipe()` и `dup()`.
- It’s all just system calls!