Turinys

L4_1

fork() – naujo proceso sukūrimas

Funkcijos aprašas:

```
pid_t fork(void);
```

Iškvietus `fork()` sukuriamas naujas vaiko procesas ir toliau OS vykdo abu šiuos procesus. Vaiko procesas yra tėvo proceso kopija, t.y. vaikas paveldi iš tėvo identišką atminties kopiją, CPU registrų turinius, daugumą atidarytų failų ir soketų, proceso kontekstą (PCB – *Process Controll Block*) ir t.t. (žr.: POSIX arba `man`). Sukuriama *copy-on-write* tėvo atminties kopija, t.y. abiejų procesų atmintis vienoda tol, kol nei tėvas, nei vaikas į ją nerašo, o kai rašo – padaroma tikra kopija (paprastai tik modifikuoto puslapio).

`fork()` grąžinamos reikšmės (`pid_t`):

- `-1` – jei vaiko proceso sukurti nepavyko (įvyko klaida);
- `0` – sukurtam vaiko procesui;
- `N` (sveikas teigiamas skaičius) – vaiko proceso PID tėvo procesui.

getpid()

```
pid_t getpid(void);
```

Funkcija `getpid()` grąžina ją iškvietusio proceso PID.

getppid()

```
pid_t getppid(void);
```

Funkcija `getppid()` grąžina ją iškvietusio proceso PPID (tėvo PID).

wait()

```
pid_t wait(int *stat_loc);
```

Funkcija `wait()` laukia, kol kuris nors funkciją iškvietusio proceso vaikų pasibaigs arba bus sustabdytas (STOP signalu). Funkcija grąžina pasibaigusio vaiko proceso PID. Jei `stat_loc` ne NULL – šiuo adresu įrašoma vaiko proceso grąžinta reikšmė ir grįžimo priežastis. Smulkiau apie grąžintos reikšmės apdorojimą (pvz. kaip iš `stat_loc` išskirti vaiko pabaigos kodą) žr. POSIX standarte ir `man` puslapiuose.

system()

```
int system(const char *command);
```

Funkcijos `system()` argumentas interpretuojamas kaip `sh` komandos eilutė, t.y. vykdoma `sh -c "command"`. Argumentas gali būti bet kokia `sh` išraiška (ne vien paleidžiamos programos vardas ir jos argumentai).

exec* funkcijos

`exec*()` funkcijų grupė leidžia pakeisti proceso kūną nauju vykdomu kodu. Šiai grupei priklauso funkcijos:

```
extern char **environ;

int execl(const char *path, const char *arg0, ... /*, (char *)0 */);

int execle(const char *path, const char *arg0, ... /*,

       (char *)0, char *const envp[]*/);

int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);

int execv(const char *path, char *const argv[]);

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

int execvp(const char *file, char *const argv[]);

int fexecve(int fd, char *const argv[], char *const envp[]);
```

Sėkmės atveju šios funkcijos **nieko negrąžina**, kadangi kviečiančio proceso kūnas perdengiamas nauju kūnu (seno kodo nebėra, todėl nėra kur grįžti).

Tarkim, norime iškviesti `ls -l dir` komandą. `execl()` iškvietimas:

```
execl( "/bin/ls", "ls", "-l", "dir", (char*)0 );
```

Funkcijų vardai sudaryti pagal šabloną `exec(l|v)p?e?`, t.y.:

- `(l|v)` – rodo, kaip iškviečiamai funkcijai perduodami paleidžiamos komandos argumentai ir aplinkos kintamieji:
  - `l` – atskirais funkcijos argumentais;
  - `v` – vienu funkcijos argumentu (masyvo adresu);
- `p` – kaip ieškoma paleidžiamos komandos (apie tai toliau bus rašoma smulkiau);
- `e` – ar funkcijos iškvietime nurodomi aplinkos kintamieji.

Jei kviečiamos funkcijos, kuriose aplinkos kintamieji nenurodomi – jie paimami iš `environ` kintamojo.

execve()

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

Argumentai:

- `path` – paleidžiamo failo (kuriuo bus pakeistas dabartinis procesas) vardas (absoliutus arba santykinis kelias);
- `argv` – argumentai, kurie paduodami paleidžiamam failui;
- `envp` – aplinkos kintamieji su jų reikšmėmis, kurie bus prieinami paleistam failui, t.y. masyve eilutės `kintamasis=reiksme`.

Paskutinis `argv` ir `envp` rodyklių masyvų elementas – NULL rodyklė.

Pagal susitarimą, pirmas `argv` sąrašo elementas – paleidžiamo failo vardas.

fork užduotys

- Išsiaiškinkite (sukurkite tai demonstruojančią programą **`loginas_fork01.c`**, nusipaišykite sukurtų procesų medį (kad matytųsi kas ką paleido)), kiek procesų sukurs programa, kurioje vykdomas fragmentas:

- P1
- ├── P2
- │ └── P4
- │
- └── P3
- └── P5

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

int main() {
    fork();
    fork();
    printf("Hello, world!\n");
    return 0;
}
```

```
• fork();

• fork();
```

- Pataisykite `loginas_fork01.c` (nusikopijuokite į **`loginas_fork01a.c`**) taip, kad būtų sukuriami 3 procesai: tėvas -> vaikas -> anūkas.

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

int main() {
    pid_t pid1, pid2;
    pid1 = fork();  // Pirmas fork()
    if (pid1 == 0) { // Vaiko procesas
```

```
        pid2 = fork(); // Antras fork()
        if (pid2 == 0) { // Anūko procesas
            printf("Anūkas (procesas %d)\n", getpid());
        } else { // Vaiko procesas tęsia darbą
            printf("Vaikas (procesas %d)\n", getpid());
        }
    } else { // Tėvo procesas tęsia darbą
        printf("Tėvas (procesas %d)\n", getpid());
    }
    return 0;
}
```

- Pataisykite `loginas_fork01.c` (nusikopijuokite į **`loginas_fork01b.c`**) taip, kad būtų sukuriami 3 procesai: tėvas ir du jo vaikai .

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

int main() {
    pid_t pid1, pid2;
    pid1 = fork();  // Pirmas fork()
    if (pid1 == 0) { // Pirmo vaiko procesas
        printf("Pirmas vaikas (procesas %d)\n", getpid());
    } else { // Tėvo procesas
        pid2 = fork(); // Antras fork()
        if (pid2 == 0) { // Antro vaiko procesas
            printf("Antras vaikas (procesas %d)\n", getpid());
        } else { // Tėvas tęsia darbą
            printf("Tėvas (procesas %d)\n", getpid());
        }
    }
    return 0;
}
```

- Išsiaiškinkite, kas tapo proceso, kurio tėvas pasibaigė, tėvu. Sukurkite tai demonstruojančią programą **`loginas_fork02.c`**.

Po to, kai tėvo procesas pasibaigia, vaiko procesas tampa "naudojančiu" procesu, kurio tėvas yra procesas su ID 1 (init procesas).

- Išsiaiškinkite, kaip procesų sąraše matomas "zombis", t.y. pasibaigęs vaiko procesas, kol tėvas su `wait()` dar nenuskaitė vaiko pabaigos būsenos? Sukurkite tai demonstruojančią programą **`loginas_fork03.c`**. Procesų sąrašui gauti galite naudoti `ps` komandą su tinkamais argumentais (jos iškvietimui galite naudoti `system()` arba `exec()` savo nuožiūra).

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

int main() {
    pid_t pid;
    pid = fork(); // Sukuriamas naujas procesas
    if (pid == -1) { // Klaida
        perror("fork");
        exit(1);
    } else if (pid == 0) { // Vaiko procesas
```

```
        printf("Vaikas (procesas %d), tėvas (procesas %d)\n", getpid(), getppid());
        sleep(10); // Palaukime 10 sekundžių
        printf("Vaikas pasibaigė (procesas %d), tėvas (procesas %d)\n", getpid(), getppid());
    } else { // Tėvo procesas
        printf("Tėvas (procesas %d), tėvas (procesas %d)\n", getpid(), getppid());
        sleep(5); // Palaukime 5 sekundes, kol vaikas pasibaigs
        system("ps -o pid,ppid,state,tty,command"); // Naudojame ps komandą, kad pamatytume vaiko procesą kaip "Z"
        wait(NULL); // Laukiame, kol vaiko procesas bus sunaikintas
        printf("Vaiko procesas sunaikintas (procesas %d)\n", pid);
    }
    return 0;
}
```

exec užduotys

- sukurkite programą **loginas_exec01.c**, kuri:
  - priimtų vieną argumentą – sveiką skaičių;
  - atspausdintų savo PID, PPID ir gautą argumentą;
  - jei argumento reikšmė >0 paleistų save pačią su `exec()`, bet nurodydama vienetu mažesnę argumento reikšmę.

```
#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>


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

   printf("Mano PID: %d\n", getpid());

   printf("Mano tėvo PID: %d\n", getppid());


   if (argc < 2) {

      printf("Nepateiktas argumentas!\n");

      return 1;

   }


   int argumentas = atoi(argv[1]);

   printf("Gautas argumentas: %d\n", argumentas);
```

```
    if (argumentas > 0) {

        printf("Paleidžiu save su argumentu %d\n", argumentas - 1);

        char buf[50];

        snprintf(buf, 50, "%d", argumentas - 1);

        execl("./lukkuz1_exec01", "./lukkuz1_exec01", buf, (char*)NULL);

        perror("execl() klaida");

        exit(1);

    }


    return 0;

}
```

- [advanced] pataisykite `loginas_exec01.c` (nusikopijuokite į **`loginas_exec01a.c`**), kad būtų naudojama `exec*p*()` funkcija. Kaip tokią programą paleisti, kad ji veiktų, kaip anksčiau?

```
#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>


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

    int num;

    pid_t pid;


    if (argc != 2) {

        fprintf(stderr, "Usage: %s number\n", argv[0]);

        exit(EXIT_FAILURE);

    }


    num = atoi(argv[1]);

    printf("PID: %d, PPID: %d, Argument: %d\n", getpid(), getppid(), num);
```

```
    if (num > 0) {

        char arg1[10];

        sprintf(arg1, "%d", num - 1);

        char* args[] = {"./lukkuz1_exec01a", arg1, NULL};

        execvp(args[0], args);

        perror("execvp");

        exit(EXIT_FAILURE);

    }



    return 0;

}
```

- sukurkite programą **loginas_exec02.c**, kuri darbiniame kataloge sukurtų sh skriptą ir jį paleistų.

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

int main() {
    char *script_name = "test.sh";
    FILE *script_file = fopen(script_name, "w");

    // Rašome skripto turinį į failą
    fprintf(script_file, "#!/bin/sh\n");
    fprintf(script_file, "echo 'Hello, world!'\n");

    // Uždarome failą
    fclose(script_file);

    pid_t pid = fork();
    if (pid == 0) {
        // Vaikas vykdys sh skriptą
        execl("/bin/sh", "sh", script_name, NULL);
        perror("execl");
        exit(EXIT_FAILURE);
    } else if (pid > 0) {
        // Tėvas laukia, kol vaikas baigs darbą
        wait(NULL);
        printf("Skriptas baigtas vykdyti.\n");
    } else {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    // Ištriname skriptą
    remove(script_name);
```

```
    return 0;
}
```

L4_2

Signalai

Signalai – tai pranešimai, kurie yra siunčiami procesui informuojant jį apie įvairius įvykius. Pagal jų veiklos scenarijų atėjus signalui tuo metu vykdomas procesas yra pertraukiamas ir reikalaujama atėjusio signalo apdorojimo. Signalai pagal savo prigimtį gali būti įvairių rūšių. Kiekvienas iš jų yra susijęs su tam tikru įvykiu ir yra pažymimas priskirtu vardu bei numeriu. Detalius signalų aprašymus galite rasti `signal()` f-ją aprašančiuose `man` puslapiuose.

Sistemos palaikomų signalų vardų sąrašą galima išsivesti naudojant komandą:

```
$ kill -l
```

Kiekvienam signalui paprastai yra numatyta tą signalą apdorojanti funkcija, kuri yra iškviečiama procesui gavus signalą. Pavyzdžiui:

- `Ctrl-C` klavišų paspaudimas priverčia sistemą pasiųsti vykdomam procesui `INT` tipo signalą (`SIGINT`). Pagal nutylėjimą šis signalas priverčia nutraukti procesą.
- `Ctrl-Z` klavišų paspaudimas priverčia sistemą pasiųsti vykdomam procesui `TSTP` signalą (`SIGTSTP`). Pagal nutylėjimą šis signalas priverčia suspenduoti procesą vykdymą.
- `Ctrl-\` klavišų paspaudimas priverčia sistemą pasiųsti vykdomam procesui `ABRT` signalą (`SIGABRT`). Pagal nutylėjimą šis signalas priverčia nutraukti procesą. Jo poveikis toks pat kaip Ctrl-C paspaudimas.
- ir t.t.

Signalai gali būti pasiunčiami iš komandinės eilutės, naudojant įvairias komandas – tai dažniausiai shell'o komandos (viena tokių - `kill` komanda, su kuria susipažinote LD14 lab. darbo metu).

Jei procesas nėra numatęs savo specialaus signalų apdorojimo, tai jam yra nustatomas signalų apdorojimas pagal nutylėjimą. Pavyzdžiui, signalo `TERM` atveju yra vykdomas sisteminis `exit()` kreipinys. Pagal nutylėjimą signalo `ABRT` atveju yra atliekamas proceso atminties turinio išvedimas į failą "core", vartotojo einamajame kataloge. Signalų apdorojimo pagal nutylėjimą aprašymus galite rasti `signal()` f-ją aprašančiuose `man` puslapiuose.

Signalų apdorojimas

Procesui leidžiama kai kuriuos signalus atidėti, ignoruoti arba specialiai apdoroti.
Signalo apdorojimui galima numatyti tam tikrus veiksmus, kurie paprastai surašomi kaip signalo apdorojimo funkcija. Signalo apdorojimas realizuojamas naudojant kreipinį `signal()`. Pilnas f-jos `signal()` aprašas:

```
void (*signal(int sig, void (*func)(int)))(int);
```

Argumentai:

- **sig** - Signalo ID (vardas)
- **func** - Rodyklė į f-ją, kuri iškviečiama signalo pasirodymo metu. Šioje f-joje aprašomi veiksmai susiję su nurodyto signalo apdorojimu.

SIGINT signalo specialaus (ne pagal nutylėjimą) apdorojimo pavyzdys (source:posix|inglagz_signal00.c):

```c
/* Ingrida Lagzdinyte-Budnike KTK inglagz */
/* Failas: inglagz_signal00.c */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

void il_catch_INT(int);

void il_catch_INT(int snum) {
   printf("Caught signal %d, coming out...\n", snum);
   exit(1);
}

int main(int argc, char **argv) {
   printf( "(C) 2013 Ingrida Lagzdinyte-Budnike, %s\n", __FILE__ );
   signal(SIGINT, il_catch_INT);

   while(1)
   {
      printf("Going to sleep for a second...\n");
      sleep(1);
   }

   return(0);
}
```

Paleidus sukompiliuotą programą, vykdomas begalinis ciklas, tačiau jo vykdymo metu <Ctrl+C> klavišų paspaudimu siunčiamas signalas apdorojamas ne pagal nutylėjimą (t.y. vietoje programos nutraukimo, išvedamas pranešimas ir tik tuomet nutraukiamas programos darbas):

```
$ ./inglagz_signal00
```

```
(C) 2013 Ingrida Lagzdinyte-Budnike, inglagz_signal00.c

Going to sleep for a second...

Going to sleep for a second...

Going to sleep for a second...

^CCaught signal 2, coming out...
```

SIGCHLD signalo specialaus (ne pagal nutylėjimą) apdorojimo pavyzdys (source:posix|inglagz_signal01.c):

```c
/* Ingrida Lagzdinyte-Budnike KTK inglagz                                    */
/* Failas: inglagz_signal01.c                                                */
/*                                                                           */
/* inglagz_signal01: tevo procesas sukuria vaiko procesa                     */
/* ir laukia vaiko proceso darbo pabaigos signalo                            */
/* Jo sulaukes darba baigia pats                                             */
/* Vaiko procesas isspausdina pranesima ir palaukes 3s baigia darba          */

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

void il_catch_CHLD(int);        /* signalo apdorojimo f-ja */
void il_child(void);               /* vaiko proceso veiksmai */
void il_parent(int pid);        /* tevo proceso veiksmai */

void il_child(void) {
    printf("        child: I'm the child\n");
    sleep(3);
    printf("        child: I'm exiting\n");
    exit(123);
}

void il_parent(int pid) {
    printf("parent: I'm the parent\n");
    sleep(10);
```

```
29      printf("parent: exiting\n");
30  }
31
32  void il_catch_CHLD(int snum) {
33      int pid;
34      int status;
35
36      pid = wait(&status);
37      printf("parent: child process (PID=%d) exited with value %d\n", pid, WEXITSTATUS(status));
38  }
39
40  int main(int argc, char **argv) {
41      int pid;                                /* proceso ID */
42
43      printf( "(C) 2013 Ingrida Lagzdinyte-Budnike, %s\n", __FILE__ );
44
45      signal(SIGCHLD, il_catch_CHLD);        /* aptikti vaiko proc pasibaigima ir apdoroti */
46      switch (pid = fork()) {
47      case 0:                                /* fork() grazina 0 vaiko procesui */
48          il_child();
49          break;
50      default:                                /* fork() grazina vaiko PID tevo procesui */
51          il_parent(pid);
52          break;
53      case -1:                                /* fork() nepavyko */
54          perror("fork");
55          exit(1);
56      }
57      exit(0);
58  }
```

Signalai gali turėti specialią paskirtį (prieš tai nagrinėti atvejai: SIGINT, SIGSYS, SIGTSTP, SIGCHLD ir t.t.) arba jų prasmę gali nusakyti pats vartotojas. Tai taikytina signalams SIGUSR1 bei SIGUSR2.

Signal	Value	Description
SIGHUP	1	Hangup (POSIX) Report that user's terminal is disconnected. Signal used to report the termination of the controlling process.
SIGINT	2	Interrupt (ANSI) Program interrupt. (ctrl-c)
SIGQUIT	3	Quit (POSIX) Terminate process and generate core dump.
SIGILL	4	Illegal Instruction (ANSI) Generally indicates that the executable file is corrupted or use of data where a pointer to a function was expected.
SIGTRAP	5	Trace trap (POSIX)
SIGABRT SIGIOT	6	Abort (ANSI) IOT trap (4.2 BSD) Process detects error and reports by calling abort
SIGBUS	7	BUS error (4.2 BSD) Indicates an access to an invalid address.
SIGFPE	8	Floating-Point arithmetic Exception (ANSI). This includes division by zero and overflow.The IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985) defines various floating-point exceptions.
SIGKILL	9	Kill, unblockable (POSIX) Cause immediate program termination. Can not be handled, blocked or ignored.
SIGUSR1	10	User-defined signal 1
SIGSEGV	11	Segmentation Violation (ANSI) Occurs when a program tries to read or write outside the memory that is allocated for it by the operating system, dereferencing a bad or NULL pointer. Indicates an invalid access to valid memory.
SIGUSR2	12	User-defined signal 2
SIGPIPE	13	Broken pipe (POSIX) Error condition like trying to write to a socket which is not connected.
SIGALRM	14	Alarm clock (POSIX) Indicates expiration of a timer. Used by the `alarm()` function.
SIGTERM	15	Termination (ANSI) This signal can be blocked, handled, and ignored. Generated by "kill" command.
SIGSTKFLT	16	Stack fault
SIGCHLD SIGCLD	17	Child status has changed (POSIX) Signal sent to parent process whenever one of its child processes terminates or stops.

Signal	Value	Description
SIGCONT	18	Continue (POSIX) Signal sent to process to make it continue.
SIGSTOP	19	Stop, unblockable (POSIX) Stop a process. This signal cannot be handled, ignored, or blocked.
SIGTSTP	20	Keyboard stop (POSIX) Interactive stop signal. This signal can be handled and ignored. (ctrl-z)
SIGTTIN	21	Background read from tty (POSIX)
SIGTTOU	22	Background write to tty (POSIX)
SIGURG	23	Urgent condition on socket (4.2 BSD) Signal sent when "urgent" or out-of-band data arrives on a socket.
SIGXCPU	24	CPU limit exceeded (4.2 BSD)
SIGXFSZ	25	File size limit exceeded (4.2 BSD)
SIGVTALRM	26	Virtual Time Alarm (4.2 BSD) Indicates expiration of a timer.
SIGPROF	27	Profiling alarm clock (4.2 BSD) Indicates expiration of a timer. Use for code profiling facilities.
SIGWINCH	28	Window size change (4.3 BSD, Sun)
SIGIO SIGPOLL	29	I/O now possible (4.2 BSD) Pollable event occurred (System V) Signal sent when file descriptor is ready to perform I/O (generated by sockets)
SIGPWR	30	Power failure restart (System V)
SIGSYS	31	Bad system call

Signalų siuntimas

Procesai gali siųsti signalus kitiems procesams ne tik iš komandinės eilutės (t.y. ne tik naudojant shell komandas). Signalai gali būti persiunčiami kitam procesui naudojant sisteminį kreipinį `kill()`. Pilnas f-jos `kill()` aprašas:

```
int kill(pid_t pid, int sig);
```

Argumentai:

- **pid** - Proceso ID
- **sig** - Signalo ID (vardas)

Signalo SIGUSR1 siuntimo ir apdorojimo pavyzdys, kai tėvo procesas pasiunčia signalą vaiko procesui (source:posix|inglagz_signal02.c):

```c
/* Ingrida Lagzdinyte-Budnike KTK inglagz */
/* Failas: inglagz_signal02.c */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <wait.h>
#include <signal.h>
#include <stdlib.h>

static int received_sig = 0;

void il_catch_USR1( int );       /* signalo apdorojimo f-ja */
int il_child( void );            /* vaiko proceso veiksmai */
int il_parent( pid_t pid );      /* tevo proceso veiksmai */

int il_child( void ){
   sleep( 1 );
   printf( "        child: my ID = %ld\n", getpid() );
   while( 1 )
      if ( received_sig == 1 ){
          printf( "        child: Received signal from parent!\n" );
          sleep( 1 );
          printf( "        child: I'm exiting\n" );
          return 0;
      }
}

int il_parent( pid_t pid ){
   printf( "parent: my ID = %ld\n", getpid() );
   printf( "parent: my child's ID = %ld\n", pid );
```

```
32    sleep( 3 );
33    kill( pid, SIGUSR1 );
34    printf( "parent: Signal was sent\n" );
35    wait( NULL );
36    printf( "parent: exiting.\n" );
37    return 0;
38 }
39
40 void il_catch_USR1( int snum ) {
41    received_sig = 1;
42 }
43
44
45 int main( int argc, char **arg ){
46
47    pid_t  pid;
48
49    printf( "(C) 2013 Ingrida Lagzdinyte-Budnike, %s\n", __FILE__ );
50
51    signal(SIGUSR1, il_catch_USR1);
52    switch( pid = fork() ){
53       case 0:                                    /* fork() grazina 0 vaiko procesui */
54          il_child();
55          break;
56       default:                                   /* fork() grazina vaiko PID tevo procesui */
57          il_parent(pid);
58          break;
59       case -1:                                   /* fork() nepavyko */
60          perror("fork");
61          exit(1);
62    }
63    exit(0);
64 }
65
```

Programiniai kanalai

Programinius kanalus galima organizuoti tarp dviejų giminingų procesų: pavyzdžiui, tarp tėvo ir vaiko proceso.

Veiksmai atliekami tokiu pačiu principu kaip ir rašymas/skaitymas į/iš failo. Programinį kanalą sukuria kreipinys **pipe(fd)**, kuris įvykdomas prieš sukuriant procesus vaikus. Kreipinio metu yra sukuriami du deskriptoriai:

- fd[0] yra skaitymo iš programinio kanalo deskriptorius,
- fd[1] - rašymo į programinį kanalą deskriptorius.

Kanalas yra naudojamas baitų srauto persiuntimui viena kryptimi, todėl norint turėti abipuses komunikacijas reiktų kurti du kanalus.
Reikia atsiminti, kad kanale naudojamas buferis yra ribotas, taigi rašant daugiau duomenų procesas bus blokuojamas, kol jau įrašyti duomenys bus nuskaityti.
Kanalas turi būti sukuriamas prieš sukuriant vaiko procesą, t.y. **pipe()** kvietinys turi eiti prieš **fork()**.
Bet kuris iš procesų gali tiek rašyti į kanalą, tiek iš jo skaityti, todėl reikia susitarti, kuris iš procesų rašys į kanalą ir kuris skaitys. Abu procesai (tėvo ir vaiko) žino abu deskriptorius, bet naudojasi vienu.
Nenaudojamas deskriptorius turėtų būti uždaromas naudojant **close()**.
Pavyzdžiui:

(source:posix|nijsara_pipe01.c)

```c
/* Nijole Sarafiniene KTK nijsara */
/* Failas: nijsara_pipe01.c */

/* Programoje yra loginė klaida. Suraskite kur. */

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

int main (){
   int      fd[2];
   pid_t    pid, x;
   int      status;
   printf( "(C) 2013 Nijole Sarafiniene, %s\n", __FILE__ );

   if( pipe( fd ) == -1 ){
      fprintf( stderr, "Nepavyko sukurti programinio kanalo !\n" );
      exit( 1 );
   }
   pid = getpid();
   if( write( fd[1], &pid, sizeof( pid ) ) != sizeof( pid ) ){
      fprintf( stderr, "Klaida rasant" );
      exit( 2 );
```

```
26    }
27    pid = fork();
28    if( pid == 0 ){
29       sleep( 1 );
30       read( fd[0], &pid, sizeof( pid ) );
31       printf ( "(vaikas) Tevo proceso ID: %ld\n", pid );
32       exit( 1 );
33    }
34    else if( pid == -1 ){
35       fprintf ( stderr, "Nepavyko sukurti vaiko !\n" );
36       exit( 4 );
37    }
38    else{
39       printf("(tevas) Mano PID: %ld\n", getpid() );
40       x = wait( &status );
41       return 0;
42    }
43 }
```

Įvardyti kanalai

Tai kanalas su specifiniu vardu. Juo gali naudotis negimininingi procesai. Pranešimą, nusiųstą į šio tipo kanalą gali skaityti bet koks autorizuotas procesas, kuris žino įvardyto kanalo vardą. Įvardyti kanalai kartais vadinami FIFO. Sukuriamas FIFO **mknod** sisteminiu kreipiniu:

```
int mknod (char *pathname, int mode, int dev)
```

POSIX API: gijos

Pthreads funkcijos- tai paprastas ir veiksmingas būdas siekiant sukurti kelių gijų taikomąją programą. Pradžioje main() programa atitinka vieną giją. Kitos gijos turi būti sukuriamos. Kintamojo tipas **pthread_t** yra nuorodos į gijas priemonė.

pthread_create()

```
pthread_create (thread, attr, start_routine, arg)
```

Ši funkcija sukuria naują giją ir padaro ją vykdomąja, visos gijos yra lygiavertės.
Gijos ID grįžta per **thread** argumentą.
start_routine- tai C kalbos f-ja, kurią vykdys sukurta gija,
arg - tai argumentas, perduodamas **start_routine** ;
attr – galima nusakyt, ar pagrindinė gija lauks vaiko veiksmų pabaigos ar tiesiog vykdys savo veiksmus, ar pagimdyta gija turi būti surišama su branduolio gija.

pthread_join()

Funkcija pthread_join – naudojama sulaukti gijos pabaigos:

```
int pthread_join(pthread_t thread, void **value_ptr)
```

Kompiliuojant programą, reikia pridėti **-lpthread** opciją. ty.:

```
 gcc program.c -o programa -lpthread
```

Programos (source:posix|nijsara_pthread01.c) pavyzdys:

```
1  /* Nijole Sarafiniene KTK nijsara */
2  /* Failas: nijsara_fredas01.c */
3
4  #include <pthread.h>
5  #include <stdio.h>
6
7  /* sia f-ja vykdys sukurta gija  */
8  void *inc_x(void *x_void_ptr){
9
10    /* didinti  x iki 100 */
11    int *x_ptr = (int *)x_void_ptr;
12    while(++(*x_ptr) < 100);
13
14    printf("x padidintas \n");
15
16    /* f-ja turi kazka grazinti- pvz NULL  */
17    return NULL;
18 }
19
20 int main(){
21    pthread_t inc_x_thread; /* tai kintamasis suristas su sukuriama gija  */
22    int x,y;
23    printf( "(C) 2013 Nijole Sarafiniene, %s\n", __FILE__ );
24    x = 0, y = 0;
25
26    /* pradines  x ir y reiksmes */
```

```
27    printf("x: %d, y: %d\n", x, y);
28
29    /* sukuriama gija, kuri vykdys  inc_x(&x) */
30    if(pthread_create(&inc_x_thread, NULL, inc_x, &x)) {
31       fprintf(stderr, "Klaida sukuriant gija \n");
32       return 1;
33    }
34
35    /* didinamas  y iki 100 0-neje --pagrindineje---gijoje */
36    while(++y < 100);
37
38    printf("y padidintas \n");
39
40    /* sulaukt kol baigsis kita gija */
41    if(pthread_join(inc_x_thread, NULL)) {
42       fprintf(stderr, "Klaida sujungiant gijas \n");
43       return 2;
44    }
45
46    /* rezultatu isvedimas - x yra  100 nes veike dvi gijos */
47    printf("x: %d, y: %d\n", x, y);
48
49    return 0;
50 }
```

Veiksmams su semaforais naudojamos funkcijos: sem_init – semaforo inicializacijai, sem_wait – semaforo užrakinimui, sem_post – semaforui atrakinti.

Užduotys

Procesų komunikacija naudojant signalus

- Komanda man peržiūrėkite kokie signalai galimi sistemoje bei koks jų apdorojimas pagal nutylėjimą.

SIG_IGN – signalas ignoruojamas.

SIG_DFL – vykdomas default veiksmas.

SIGKILL – signal nutraukimas.

SIGSTOP – signal nutraukimas.

- Modifikuokite programą (source:posix|inglagz_signal01.c) pavadindami ją `loginas_signal01a.c` taip, kad vaiko procesas prieš baigdamas savo darbą išsiųstų tėvo procesui alarmo signalą `SIGALRM`, o šis gavęs šį signalą jį apdorotų išvesdamas pranešimą, kad signalą gavo (nurodant jo numerį).

```
#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

#include <stdio.h>

#include <sys/wait.h>



void il_catch_CHLD(int);        /* signalo apdorojimo f-ja */

void il_catch_ALRM(int);        /* signalo apdorojimo f-ja */

void il_child(void);            /* vaiko proceso veiksmai */

void il_parent(int pid);        /* tevo proceso veiksmai */



void il_child(void) {

   printf("        child: I'm the child\n");

   sleep(3);

   printf("        child: Sending SIGALRM signal to parent\n");

   kill(getppid(), SIGALRM);

   exit(123);

}



void il_parent(int pid) {

   printf("parent: I'm the parent\n");
```

```
    sleep(10);
    printf("parent: Exiting\n");
}


void il_catch_CHLD(int snum) {
    int pid;
    int status;

    pid = wait(&status);
    printf("parent: Child process (PID=%d) exited with value %d\n", pid, WEXITSTATUS(status));
}


void il_catch_ALRM(int snum) {
    printf("parent: Received SIGALRM signal (%d)\n", snum);
}


int main(int argc, char **argv) {
    int pid;    /* proceso ID */

    printf("(C) 2013 Ingrida Lagzdinyte-Budnike, %s\n", __FILE__);

    signal(SIGCHLD, il_catch_CHLD);    /* aptikti vaiko proc pasibaigima ir apdoroti */
    signal(SIGALRM, il_catch_ALRM);    /* aptikti SIGALRM signalą ir apdoroti */

    switch (pid = fork()) {
        case 0:    /* fork() grazina 0 vaiko procesui */
```

```
            il_child();

            break;

        default:    /* fork() grazina vaiko PID tevo procesui */

            il_parent(pid);

            break;

        case -1:    /* fork() nepavyko */

            perror("fork");

            exit(1);

    }


    exit(0);

}
```

- Modifikuokite programą (source:posix|inglagz_signal02.c) pavadindami `loginas_signal02a.c` taip, kad:
  - vaiko procesas, gavęs signalą `SIGUSR1` iš tėvo proceso įvykdo komandą `who` ir siunčia tėvo procesui signalą `SIGUSR2`.
  - tėvo procesas, gavęs iš vaiko proceso signalą, vaiko procesą "nužudo", palaukia 5 sekundes, išspausdina pranešimą apie darbo pabaigą ir baigiasi.

```
#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <wait.h>

#include <signal.h>

#include <stdlib.h>




static int received_sig = 0;




void il_catch_USR1(int);        /* signalo apdorojimo f-ja */

void il_catch_USR2(int);        /* signalo apdorojimo f-ja */
```

```
int il_child(void);             /* vaiko proceso veiksmai */
int il_parent(pid_t pid);       /* tevo proceso veiksmai */



int il_child(void) {
  sleep(1);
  printf("        child: my ID = %ld\n", (long)getpid());
  while (1) {
    if (received_sig == 1) {
      printf("        child: Received signal SIGUSR1 from parent!\n");
      printf("        child: Executing 'who' command:\n");
      system("who"); // Įvykdo komandą "who"
      sleep(1);
      printf("        child: Sending SIGUSR2 signal to parent\n");
      kill(getppid(), SIGUSR2); // Siunčia SIGUSR2 signalą tėvo procesui
      printf("        child: I'm exiting\n");
      return 0;
    }
  }
}



int il_parent(pid_t pid) {
  printf("parent: my ID = %ld\n", (long)getpid());
  printf("parent: my child's ID = %ld\n", (long)pid);
  sleep(3);
  kill(pid, SIGUSR1); // Siunčia SIGUSR1 signalą vaiko procesui
  printf("parent: Signal SIGUSR1 was sent to child\n");
  signal(SIGUSR2, il_catch_USR2); // Aptinka SIGUSR2 signalą iš vaiko proceso
```

```
  pause(); // Laukia, kol gaus SIGUSR2 signalą
  printf("parent: Child process terminated\n");
  sleep(5);
  printf("parent: Exiting\n");
  return 0;
}



void il_catch_USR1(int snum) {
  received_sig = 1;
}



void il_catch_USR2(int snum) {
  printf("parent: Received SIGUSR2 signal (%d) from child\n", snum);
}



int main(int argc, char **arg) {
  pid_t pid;
  printf("(C) 2013 Ingrida Lagzdinyte-Budnike, %s\n", __FILE__);
  switch (pid = fork()) {
    case 0:                     /* fork() grazina 0 vaiko procesui */
      il_child();
      break;
    default:                    /* fork() grazina vaiko PID tevo procesui */
      il_parent(pid);
      break;
    case -1:                     /* fork() nepavyko */
```

```
        perror("fork");
        exit(1);
    }
    exit(0);
}
```

Procesų komunikacija naudojant programinius kanalus

- Sukurkite programą loginas_pipe02.c, kuri paleistų du procesus ir iš tėvo vaikui persiųstų komandinės eilutės argumentu nurodomo failo turinį (panaudojant `pipe()`).

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



int main(int argc, char *argv[]) {
    int fd[2]; // Pipe deskriptoriai
    pid_t pid;

    if (argc < 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        exit(1);
    }

    if (pipe(fd) == -1) {
        perror("pipe");
        exit(1);
    }
```

```
pid = fork();


if (pid == -1) {
    perror("fork");
    exit(1);
} else if (pid == 0) {
    // Vaiko procesas

    close(fd[1]); // Uždarome rašymo galą, nes vaiko procesui reikia tik skaitymo

    char buffer[1024];
    int nbytes = read(fd[0], buffer, sizeof(buffer));
    printf("Child received data from parent: %.*s\n", nbytes, buffer);

    close(fd[0]); // Uždarome skaitymo galą

    printf("Child process exiting.\n");
    exit(0);
} else {
    // Tėvo procesas

    close(fd[0]); // Uždarome skaitymo galą, nes tėvo procesui reikia tik rašymo

    FILE *file = fopen(argv[1], "r");
    if (file == NULL) {
        perror("fopen");
        exit(1);
    }
```

```
        char buffer[1024];

        int nbytes;

        while ((nbytes = fread(buffer, 1, sizeof(buffer), file)) > 0) {

            write(fd[1], buffer, nbytes);

        }


        fclose(file);


        close(fd[1]); // Uždarome rašymo galą


        wait(NULL); // Laukiame vaiko proceso pabaigos


        printf("Parent process exiting.\n");

        exit(0);

    }

}
```

Gijos

- Išsiaiškinkite, sukompiliuokite ir įvykdykite fredai.c programą, esančią kataloge `/data/ld/ld4`.

```
lukkuz1@oslinux ~/lab4_2 $ ./fredai

main thread -- Hi.  I'm thread 815711808

Hi.  I'm thread 840889920

Hi.  I'm thread 832497216

Hi.  I'm thread 840894272

I'm 840894272 Trying to join with thread 840889920

840894272 Joined with thread 840889920

I'm 840894272 Trying to join with thread 832497216

840894272 Joined with thread 832497216

I'm 840894272 Trying to join with thread 824104512
```

```
Hi.  I'm thread 824104512

840894272 Joined with thread 824104512

I'm 840894272 Trying to join with thread 815711808

840894272 Joined with thread 815711808
```

```
/*
 * CS170:
 * print4.c -- forks off THREADS threads that print their ids
 */

#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#define THREADS 4

int Ego()
{
    int return_val;

    return_val = (int)pthread_self();

    return(return_val);
}

void *printme(void *arg)
{
```

```
    printf("Hi.  I'm thread %d\n", Ego());
    return NULL;
}

int
main()
{
    int i;
    int *vals;
    pthread_t *tid_array;
    void *retval;
    int err;

    vals = (int *)malloc(THREADS*sizeof(int));
    if(vals == NULL)
    {
        exit(1);
    }

    tid_array = (pthread_t *)malloc(THREADS*sizeof(pthread_t));
    if(tid_array == NULL)
    {
        exit(1);
    }

    for (i = 0; i < THREADS; i++)
    {
        err = pthread_create(&(tid_array[i]),
                             NULL,
```

```
                        printme,
                        NULL);

        if(err != 0)
        {
                fprintf(stderr,"thread %d ",i);
                perror("on create");
                exit(1);
        }

}

printf("main thread -- ");
printme(NULL);          /* main thread */


for (i = 0; i < THREADS; i++)
{

        printf("I'm %d Trying to join with thread %d\n",
                Ego(),(int)tid_array[i]);
        pthread_join(tid_array[i], &retval);
        printf("%d Joined with thread %d\n",
                Ego(),(int)tid_array[i]);
}

free(tid_array);

return(0);
```

```
    }
```

- Modifikuokite `fredai.c` programą į `fredai_skaiciuoja.c`, kurioje kiekviena gija suskaičiuotų faktorialą iki savo numerio ir jo reikšmę spausdintų.

```
#include <unistd.h>

#include <stdlib.h>

#include <pthread.h>

#include <stdio.h>

#include <sys/types.h>


#define THREADS 4


int Ego() {

   return (int)pthread_self();

}


void* calculateFactorial(void* arg) {

   int number = *((int*)arg);

   int factorial = 1;


   for (int i = 1; i <= number; i++) {

      factorial *= i;

   }


   printf("Thread %d: Factorial of %d is %d\n", Ego(), number, factorial);


   return NULL;

}
```

```
int main() {
    int i;
    int* numbers;
    pthread_t* tid_array;

    numbers = (int*)malloc(THREADS * sizeof(int));
    if (numbers == NULL) {
        exit(1);
    }

    tid_array = (pthread_t*)malloc(THREADS * sizeof(pthread_t));
    if (tid_array == NULL) {
        exit(1);
    }

    for (i = 0; i < THREADS; i++) {
        numbers[i] = i + 1;

        if (pthread_create(&(tid_array[i]), NULL, calculateFactorial, &(numbers[i])) != 0) {
            fprintf(stderr, "Thread %d: Error creating thread\n", i);
            exit(1);
        }
    }

    for (i = 0; i < THREADS; i++) {
        if (pthread_join(tid_array[i], NULL) != 0) {
            fprintf(stderr, "Thread %d: Error joining thread\n", i);
            exit(1);
```

```
        }

    }


    free(tid_array);

    free(numbers);


    return 0;

}
```

- Išsiaiškinkite, sukompiliuokite ir įvykdykite „vartotojo-gamintojo“ principais veikiančią `prod_cons.c`, esančią `/data/ld/ld4` kataloge.

Sefamoru pavyzdis, kaip juos naudoti naudojant gijas.

Testui

fd[0] – skaitymo iš kanalo deskriptorius.

fd[1] – rašymo į kanalą deskriptorius.

Fork() komandos vykdymas – komandą yra vykdoma 2^n kartų.

Getpid() – proceso pid

Getppid() – tėvo pid

Kill(int pid, signal) , jeigu pid yra teigiamas, tada siunčiamas signalas nurodytam pid.

Fork() komanda sukuriami vaikai, kai

- `-1` – jei vaiko proceso sukurti nepavyko (įvyko klaida);
- `0` – sukurtam vaiko procesui;
- `N` (sveikas teigiamas skaičius) – vaiko proceso PID tėvo procesui.

Jeigu norime perduoti savo PID kitam procesui, galime naudoti:

Pipe komandas;

Tikriname tam tikras fork komandos sąlygas.

Pthread_join – komanda kuri sujungia visas gijas joms baigus darbus.

Jeigu norime sugeneruoti antrą lygiagretų procesą vaiką, galime tai padaryti prieš wait() komandas. Tikrindami mūsų grąžinama fork komandos rezultatą.

Signalą kitiem procesam galime perduoti naudodami

Signal()

Signal(signalas, funkcija)

Signalas – koks signalas iššaukiamas, kokia funkciją vykdoma kai šitas signalas yra iššaukiamas.

Basho komandas C kalboje galime naudoti system() funkcijos pagalba,

Pavyzdžiui system(„who“) – iškviečia basho who metodą.

Jeigu norime sukūrti bendravimo ryšį tarp tėvo ir vaiko, pirma turime inicializuoti pipe kanalo fd[] masyvą ir tik po to naudoti fork() komandas.

Pthread_create(pointeris į giją, atributai, start_routine(funkcija), argumentai)

Kill(pid proceso, signalas(SIGKILL, SIGTERM...)) – komanda.