Laboratorul 1

Linia de comandă și execuția

1 Familiarizarea cu terminalul

În Tabelul 1 găsiți câteva comenzi utile navigării în terminal. Pentru fiecare citiți bine prima parte din manual. De exemplu pentru comanda `cp(1)` manualul începe cu

```
CP(1)                General Commands Manual                CP(1)

NAME
     cp - copy files

SYNOPSIS
     cp [-fipv] [-R [-H | -L | -P]] source target
     cp [-fipv] [-R [-H | -L | -P]] source ... directory

DESCRIPTION
     In the first synopsis form, the cp utility copies the
     contents of the source file to the target file.
```

Prima linie ne spune că ne aflăm în secțiunea `(1)` numită `General Commands Manual`. Ce urmează după numele comenzii în paranteză specifică secțiunea. Mai departe avem rezumatul comenzii și tipul de parametrii pe care-i acceptă. După introducere urmează descrierea pe larg. Pentru a vedea o comandă dintr-o secțiune anume folosiți

```
$ man 1 write
$ man 2 write
```

prima comandă descrie `write(1)` iar a doua syscall-ul `write(2)`.

Câteva operații și simboluri folosite în linia de comandă sunt descrise în Tabelul 2.

Urmează o sesiune exemplu în terminal unde folosim câteva din comenzile și operațiile descrise mai sus.

Comandă	Descriere
`man command`	manualul de utilizare
`pwd`	directorul curent
`ls`	conținutul directorului curent
`cp source target`	copiere fișiere
`mv source target`	mutare fișiere
`rm item`	ștergere fișiere
`mkdir dir`	creeare director
`rmdir dir`	ștergere director gol
`echo str`	repetare string
`cd path`	schimbă directorul curent

Tabela 1: Comenzi uzuale

Simbol	Descriere
`.`	directorul curent
`..`	directorul părinte
	acasă (`/home/souser`)
`cmd > file`	redirecționare ieșire către fișier
`cmd1 | cmd2`	pipe: legătură ieșire-intrare
`^`	tasta `ctrl`
`^w`	tastat concomitent `ctrl+w`

Tabela 2: Simboluri și operații în terminal

```
$ pwd
/home/souser
$ touch foo
$ ls
foo
$ cp foo bar
$ ls
bar foo
$ mv bar baz
$ ls
baz foo
$ rm baz
$ ls
foo
$ mkdir test
$ cd test/
$ pwd
/home/souser/test
$ cd ..
$ rmdir test
$ echo hello
```

```
hello
$ echo hello > hello.txt
$ cat hello.txt
hello
```

2 Analizarea execuției unui binar

Pentru a observa cum încarcă sistemul de operare un executabil de pe disk, în cele ce urmează vom crea un executabil simplu de tip helloworld.

```
#include <stdio.h>

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

Acest program se poate scrie cu ajutorul oricărui editor. În linia de comandă se pot folosi `nano(1)`, `vi(1)`, `emacs(1)` sau oricare alt editor. Subiectul acesta nu face obiectul laboratorului.

2.1 Funcții sistem

Mai deaprte, analizăm funcțiile de sistem (syscall) folosite pentru a duce executa binarul

```
$ gcc helloworld.c -o hello
$ ./hello
Hello, World!
$ ktrace hello
Hello, World!
$ kdump
```

Comanda `ktrace(1)`, scurt de la kernel trace, ne ajută să vedem de ce funcții de sistem are nevoie `hello` pentru a fii executat. Echivalentul în Linux este `strace(1)`. `kdump(1)` ne ajută să vedem fișierul binar creat de `ktrace(1)`. Pornirea execuției are loc prin apelul la `execve(2)`

```
46707 ktrace   CALL  execve(0x7f7ffffd5f08,0x7f7ffffd5da0,0x7f7ffffd5db0)
46707 ktrace   NAMI  "./hello"
46707 ktrace   ARGS
       [0] = "./hello"
41281 hello    NAMI  "/usr/libexec/ld.so"
41281 hello    RET   execve 0
```

valorile hexazecimale sunt adrese în memorie corespunzătoare argumentelor apelului de sistem. Pentru a le descifra, folosiți manualul (`$ man 2 execve`).

```
EXECVE(2)                System Calls Manual                EXECVE(2)

NAME
     execve - execute a file

SYNOPSIS
     #include <unistd.h>

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

Părțile direct influențate de programul nostru sunt

```
41281 hello     CALL  write(1,0x7bd390d0000,0xe)
41281 hello     GIO   fd 1 wrote 14 bytes
      "Hello, World!
      "
41281 hello     RET   write 14/0xe
```

unde mesajul este afișat pe ecran cu ajutorul syscall-ului `write(2)`. Conform `$ man 2 execve`

```
WRITE(2)                 System Calls Manual                WRITE(2)

NAME
     write, writev, pwrite, pwritev - write output

SYNOPSIS
     #include <unistd.h>

     ssize_t
     write(int d, const void *buf, size_t nbytes);
     [...]
DESCRIPTION
     write() attempts to write nbytes of data to the object
     referenced by the descriptor d from the buffer pointed
     to by buf.
```

Descriptorii sunt folosiți pentru a indica către anumite fișiere din sistem. Înloc de a folosi `/dir/subdir/file.txt` putem scurta cu un număr care știm că atunci când apare se referă la acest fișier (structură de tip cheie–valoare). Primii trei descriptori sunt rezervați. Detalii în Tabelul 3.

2.2 Biblioteci

Funcția `printf(3)` folosită în `helloworld.c` este implementată în biblioteca standard de C numită `libc`. Cu ajutorul utilitarului `ldd(1)` putem vedea de ce biblioteci are nevoie `hello` pentru a fii executat

Descriptor	Fișier	Folosit de
0	`stdin`	`scanf(3)`
1	`stdout`	`printf(3)`
2	`stderr`	`perror(3)`

Tabela 3: Descriptori rezervați

```
$ ldd ./hello
./hello:
Start            End              Type Open Ref GrpRef Name
9e7c6e00000      9e7c7002000      exe  1    0   0      ./hello
9ea8816b000      9ea8844a000      rlib 0    1   0      /usr/lib/libc.so.90.0
9ea60800000      9ea60800000      rtld 0    1   0      /usr/libexec/ld.so
```

Primele coloane indică unde în memorie poate fii găsită fiecare bibliotecă. Putem vedea că pe lângă `libc` mai este necesar și `ld.so`. Aceasta din urmă este o bibliotecă specifică cu care sistemul de operare caută, găsește și încarcă în memorie bibliotecile utilizate de executabil (în cazul nostru doar `libc`).

2.3 Simboluri

Pentru a vedea de ce simboluri are nevoie `hello`, putem folosi `nm(1)`

```
$ nm ./hello
00200e50 a _DYNAMIC
00200fb0 a _GLOBAL_OFFSET_TABLE_
         W _Jv_RegisterClasses
00201000 A __bss_start
00201000 A __data_start
00201000 B __dso_handle
00000550 T __fini
00200e40 D __guard_local
00000350 T __init
00000420 W __register_frame_info
000003c0 T __start
         U _csu_finish
00201000 A _edata
00201058 A _end
000003c0 T _start
         U atexit
         U exit
00000000 F helloworld.c
00000528 T main
         U puts
```

Prima coloană arată adresa la care se găsește fiecare simbol. A doua indică tipul simbolului (`U` este prescurtarea de la `Undefined`) iar ultima numele. Executa-

Comandă	Descriere
`b symbol`	oprirea execuției la simbol
`p var`	tipărește valoarea variabilei
`n`	următoarea instrucțiune
`c`	continuarea execuției
`q`	ieșire

Tabela 4: Comenzi `gdb(1)`

bilul nostru folosește explicit doar `printf(3)` din biblioteca standard care în exemplul de sus este implementat cu ajutorul lui `puts(3)` iar adresa la care se v-a găsi această funcție va fii stabilită când se va încărca în memorie biblioteca C în timpul execuției (vezi secțiunea precedentă). Un alt simbol cunoscut este `main` și `helloworld.c` (fișierul sursă).

2.4 Instrumentare

Un prim utilitar folosit la investigarea și instrumentarea executabilelor este `gdb(1)`. Întâi vom recompila cu simboluri de debug

```
$ gcc -g -O0 helloworld.c -o hello
```

Opțiunile în plus sunt `-g` care adaugă efectiv simbolurile (numărul liniei în fișierul sursă, instrucțiunea C etc.) și `-O0` pentru a elimina optimizările compilatorului ce ar putea rezulta în eliminarea anumitor variabile sau instrucțiuni C pierzându-se astfel corespondența cu fișierul sursă.

În Tabelul 4 sunt puse câteva instrucțiuni uzuale folosite pentru a depana cu `gdb`. încheiem cu o sesiune de depanare unde se observă faptul că breakpoint-ul se pune de fapt la o adresă în memorie deși argumentul este un simbol și liniile din fișierul sursă sunt afișate corespunzător.

```
$ gdb ./hello
(gdb) b main
Breakpoint 1 at 0x530: file helloworld.c, line 5.
(gdb) r
Starting program: /home/souser/hello
Breakpoint 1 at 0xbac61500530: file helloworld.c, line 5.

Breakpoint 1, main () at helloworld.c:5
5               printf("Hello, World!\n");
(gdb) n
Hello, World!
6               return 0;
(gdb) n
7       }
(gdb) n
0x00000bac61500414 in _start () from /home/souser/hello
```

```
(gdb) n
Single stepping until exit from function _start,
which has no line number information.

Program exited normally.
(gdb) q
```

Laboratorul 2

Funcții sistem

1 Utilizarea funcțiilor sistem

Reamintim faptul că funcțiile de sistem (syscalls) sunt definite în secțiunea 2 a manualului sistemului de operare. Aprofundați definiția și utilizarea fiecărei funcții folosite în acest material prin apeluri de tipul

```
$ man 2 <syscall>
```

Funcțiile cele mai des întâlnite pentru manipularea fișierelor sunt `read(2)`, `write(2)`, `stat(2)`, `open(2)`, și `close(2)`.

1.1 Citire și scriere

Am văzut în Laboratorul 1 cum se comportă `write(2)`. Similar, `read(2)` citește dintr-un descriptor `d` în buferul `buf` un număr dat de `nbytes`.

```
ssize_t read(int d, void *buf, size_t nbytes);
```

Când este executată cu succes, ieșirea funcției este numărul de bytes citiți.

1.2 Accesarea fișierelor

Pentru a obține un descriptor asociat unui fișier trebuie folosită funcția `open(2)` care deschide fișierul găsit în `path` pentru scriere și/sau citire.

```
int open(const char *path, int flags, ...);
```

Ieșirea funcției este descriptorul asociat. Modul în care va fi manipulat fișierul este dat de argumentul `flags` similar funcției standard C `fopen(3)`.

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.

Dacă fișierul cerut nu există în sistem, se poate cere creearea lui prin adăugarea flagului `O_CREAT` la cele de scriere sau citire. În acest caz, trebuie specificate și drepturile de acces la fișier în al treilea argument.

```
#define S_IRWXU 0000700     /* RWX mask for owner */
#define S_IRUSR 0000400     /* R for owner */
#define S_IWUSR 0000200     /* W for owner */
#define S_IXUSR 0000100     /* X for owner */
```

Vezi manualul `open(2)` și tabelul din `chmod(2)` pentru mai multe detalii.

Orice fișier deschis cu `open(2)` trebuie închis când nu mai este folosit cu `close(2)`.

1.3 Informații despre fișiere

Pentru a afla detlaii despre obiectele manipulate precum dimensiunea ocupată pe disk, permisiunile de acces, data la care a fost creat și modificat ultima dată, se folosește funcția `stat(2)`.

```
int stat(const char *path, struct stat *sb);
```

În câmpurile structurii de date `stat` vor fii populate informațiile de mai sus împreună cu alte detalii.

```
struct stat {
 dev_t     st_dev;     /* inode's device */
 ino_t     st_ino;     /* inode's number */
 mode_t    st_mode;    /* inode protection mode */
 nlink_t   st_nlink;   /* number of hard links */
 uid_t     st_uid;     /* user ID of the file's owner */
 gid_t     st_gid;     /* group ID of the file's group */
 dev_t     st_rdev;    /* device type */
 struct timespec st_atim;  /* time of last access */
 struct timespec st_mtim;  /* last data modification */
 struct timespec st_ctim;  /* last file status change */
 off_t     st_size;    /* file size, in bytes */
 blkcnt_t  st_blocks;  /* blocks allocated for file */
 blksize_t st_blksize;/* optimal blocksize for I/O */
 u_int32_t st_flags;   /* user defined flags for file */
 u_int32_t st_gen;     /* file generation number */
 };
```

Următorul fragment de program afișează dimensiunea fișierului `foo`.

```
#include <sys/stat.h>
...
struct stat sb;
if (stat("foo", &sb)) {
        perror("foo");
```

errno	Valoare	Descriere
1	EPERM	operația nu este permisă
2	ENOENT	fișier sau director inexistent
5	EIO	eroare de comunicare intrare/ieșire (cu un dispozitiv)
9	EBADF	descriptor inexistent
12	ENOMEM	memorie insuficientă
13	EACESS	nu sunt permisiuni suficiente de acces
14	EFAULT	adresă invalidă
22	EINVAL	argument invalid

Tabela 1: Coduri de eroare uzuale

```
        return errno;
}
printf("Foo takes %jd bytes on disk\n", sb.st_size);
```

2 Tratarea erorilor

În manualele de utilizare există o secțiune importantă numită `RETURN VALUES`. Adesea valoarea la ieșirea cu succes este pozitivă, iar când apelul întâmpină o problemă utilizatorul este semnalat prin valoarea -1. În acest caz mai multe detalii se pot găsi în variabila globală `ernno`. Codul de eroare indicat are asociat un mesaj de eroare ce poate fii ușor afișat pe ecran cu ajutorul funcției `perror(3)`.

Funcția `read(2)` spune următoarele în documentație

> **RETURN VALUES**
> If successful, the number of bytes actually read is returned. Upon reading end-of-file, zero is returned. Otherwise, a -1 is returned and the global variable errno is set to indicate the error.

deci un apel corect al funcției arată astfel

```
nread = read(fd, buf, bufsz);
if (nread < 0) {
        perror("read buf");
        return errno;
}
```

în anumite cazuri se poate face un caz special și pentru `nread == 0` semnalând că am ajuns la sfârșitul fișierului.

În Tabelul 1 puteți găsi câteva din cele mai frecvente erori semnalate de errno. O listă completă cu valorile posibile și semnificația lor se găsește în manual `errno(2)`.

Toate apelurile de funcții trebuie verificate corespunzător pentru toate ieșirile posibile, fie cu succes, fie fără!

3 Sarcini de laborator

1. Rescrieți programul HelloWorld de data trecută folosind numai funcții sistem.

2. Scrieți un program `mycp` care să primească la intrare în primul argument un fișier sursă pe care să-l copieze într-un alt fișier cu numele primit în al doilea argument. Exemplu apel `./mycp foo bar`.

Laboratorul 3

Implementare funcții sistem

1 Compilare kernel

Kernelul OpenBSD se găsește în `/bsd` iar codul sursă din care este compilat în `/sys`. Pentru a compila un kernel nou trebuie executate următoarele comenzi:

```
# cd /sys/arch/$(machine)/compile/GENERIC.MP
# make obj
# make config
# make
```

Înainte de a instala un kernel nou e bine să faceți o copie a originalului pentru a putea reveni în cazul în care noul kernel are un defect la pornirea sistemului de operare

```
# cp /bsd /bsd.1
```

după acest pas de siguranță, executați

```
# make install
```

și noul kernel va fii încărcat implicit la repornirea calculatorului

```
# reboot
```

2 Adăugarea unei noi funcții sistem

În OpenBSD funcțiile de sistem sunt definite în `/sys/kern/syscalls.master`. Din acest fișier se vor genera fișiere C care definesc structurile de date, varibilele și funcțiile necesare.

De exemplu pentru funcțiile cunoscute `read(2)` și `write(2)` veți găsi în acest fișier următoarele intrări

```
3   STD   { ssize_t sys_read(int fd, void *buf, size_t nbyte); }
4   STD   { ssize_t sys_write(int fd, const void *buf, \\
                              size_t nbyte); }
```

Primul câmp reprezintă numărul de identificare a funcției de sistem, al doilea tipul (în general noi vom folosi tot timpul funcții de sistem standard `STD`) iar ultimul câmp este definiția C a funcției prefixată cu `sys_`.

2.1 Declararea

Adăugarea unei noi intrări se face la sfârșitul fișierului `/sys/kern/syscalls.master`. ID-ul pentru funcția noastră va fi următoarea valoare după cea a ultimei funcții existente. De exemplu dacă ultimul ID este 330, noi vom folosi 331 pentru noua intrare.

```
331   STD   { int sys_khello(const char *msg); }
```

După modificarea fișierului `/sys/kern/syscalls.master` regenerați fișierele C aferente prin comanda

```
# cd /sys/kern && make syscalls
```

Fișierele generate sunt în directorul `/sys/kern` și `/sys/sys`. Modificările principale sunt

- `/sys/kern/syscalls.c` – adăugarea denumirii funcției în tabela `syscallnames`
- `/sys/sys/syscallargs.h` – definiția structurii ce va ține argumentele

  ```
  struct sys_khello_args {
         syscallarg(const char *) msg;
  };
  ```

- `/sys/sys/syscall.h` – definirea noului ID 331

  ```
  #define SYS_khello 331
  ```

  Declarația funcției are loc tot în `/sys/sys/syscallargs.h`

```
int sys_khello(struct proc *p, void *v, register_t *retval);
```

iar argumentele reale (din perspectiva kernelului) sunt

- `struct proc *p` – procesul care apelează
- `void *v` – pointer către structura `sys_khello_args`
- `register_t *retval` – pointer către rezultatul (ieșirea) funcției

Funcție	Apel	Descriere
`copyin(9)`	`copyin(ubuf, kbuf, len)`	copiază buffer user → kernel
`copyout(9)`	`copyout(kbuf, ubuf, len)`	copiază buffer kernel → user
`kcopy(9)`	`kcopy(srckbuf, dstkbuf, len)`	copiază buffer kernel → kernel
`copyinstr(9)`	`copyinstr(ubuf, kbuf, len, &done)`	copiază string user → kernel
`copyoutstr(9)`	`copyoutstr(kbuf, ubuf, len, &done)`	copiază string kernel → user
`copystr(9)`	`copystr(kbuf, kbuf, len, &done)`	copiază string kernel → kernel

Tabela 1: Funcții de copiere pentru kernel

2.2 Definirea

Funcția de sistem se definește de regulă în `/sys/kern/sys_generic.c`.

```
/*
 * Hello system call
 */
int
sys_khello(struct proc *p, void *v, register_t *retval)
{
        return 0;
}
```

Atenție, această funcție întoarce un `int` care conține un cod de eroare de tipul errno folosit mai departe de kernel. Valoarea pe care o întoarce în userland este diferită și trebuie pusă în argumentul `retval`.

Următorul pas este să citim argumentele de la intrare de la adresa indicată de `v`. Pentru asta trebuie să folosim structura `sys_khello_args` definită mai devreme.

```
        struct sys_khello_args *uap = v;
```

Conținutul structurii este comentat pentru ușurința programatorului, dar poate fi omis în funcție de gust.

Pentru a citi un argument se folosește macroul `SCARG`

```
#if _BYTE_ORDER == _BIG_ENDIAN
#define SCARG(p, k) ((p)->k.be.datum)
#elif _BYTE_ORDER == _LITTLE_ENDIAN
#define SCARG(p, k) ((p)->k.le.datum)
#else
#error "what byte order is this machine?"
#endif
```

care se ocupă cu încărcarea din registru care conține adresa la care se află structura cu argumentele trimise de utilizator (vezi Cursul 2). De exemplu, pentru a obține argumentul `msg` al noului nostru syscall `khello` folosim `SCARG(uap, msg)`.

Când avem de a face cu buffere primite din userland trebuie să le mutăm din spațiul de adresare specific procesului `p` în spațiul de adresare al kernelului. Pentru asta se pun la dispoziție seria de funcții din Tabelul 1. Mecanismul trebuie folosit în cadrul funcției de sistem `khello` pentru a copia primii 100 bytes din mesajul `msg` în spațiul kernelului

```
copyinstr(SCARG(uap, msg), kmsg, 100, NULL);
```

partea de verificare a apelului este intenționat omisă pentru simplitate. În mod normal ea trebuie să existe, dar nu face obiectul laboratorului.

3 Funcții utilitare în kernel

Atenție, în kernel nu există biblioteca C standard sau alte funcții utilitare cu care suntem obișnuiți când scriem programe în userland. Cu toate astea, în kernelul de OpenBSD, există funcții similare cu cele din standardul de C.

`printf(9)` care se comportă similar cu funcția standard `printf(3)`. Diferența este că mesajul `printf(9)` apare în consola principală (`ttyC0`) și în logul `/var/log/messages`.

Pentru alocarea și eliberarea memoriei folosiți `malloc(9)` și `free(9)`. `malloc(9)` primește două argumente în plus: tipul memoriei alocat și cum să se efectueze alocarea. De exemplu pentru a aloca 10 bytes pentru un buffer temporar (folosit doar local în funcție) se folosește apelul

```
buf = malloc(10, M_TEMP, M_WAITOK);
```

ultimul argument anunțând că apelantul poate aștepta până se găsește memorie disponibilă. Când operațiile asupra lui `buf` s-au încheiat, acesta trebuie eliberat

```
free(buf, M_TEMP, 10);
```

4 Apelare din userland

Cel mai rapid mod de a apela o nouă funcție de sistem creată este cu ajutorul lui `syscall(2)`. Această funcție de sistem apelează o altă funcție de sistem cu ID-ul din primul argument. Restul argumentelor primite sunt pasate mai departe. De exemplu, pentru a apela noua funcție `khello` cu ID-ul 331 am folosi

```
syscall(331, "foo");
```

iar pentru a apela scrie `"Hello!"` pe ecran cu ajutorul funcției cunoscute `write(2)` am apela astfel

```
syscall(4, 1, "Hello!", 6);
```

unde 4 este ID-ul lui `write(2)` conform fișierului `/sys/kern/syscalls.master`.

Apelarea elegantă cu numele funcției de sistem se face prin declararea funcției în mai multe fișiere de tip `include` din sistem. Acest pas este lăsat drept exercițiu pentru acasă.

5 Sarcini de laborator

1. Compilați un kernel nou.
2. Adăugați o funcție de sistem nouă simplă care să afișeze ceva pe ecran și demonstrați că merge apelând-o dintr-un program. Exemplu apel `syscall(id-functie, "world")`. **Atenție**, trebuie să recompilați kernelul și să reporniți sistemul de operare cu kernelul nou!
3. Modificați funcția de sistem de mai devreme să copieze un număr dat de bytes dintr-un buffer sursă într-altul destinație. La ieșire funcția va scrie numărul de bytes copiat efectiv. Verificați intrările primite și semnalați eventualele erori. Exemplu apel `sz = kcp(src,dst,10)`.

Laboratorul 4

Procese

1 Crearea unui proces nou

În mediile de dezvoltare UNIX funcția sistem cu care se creează procese noi este funcția `fork(2)`. O dată invocată, funcția creează un proces nou (numit proces fiu sau proces copil) acesta fiind o copie a procesului apelant cu câteva excepții. Indicăm aici pe cele mai importante, ele pot diferi de la sistem de operare la sistem de operare

- procesul fiu are un ID unic (denumit și `pid`)
- procesul fiu are un părinte diferit
- procesul fiu are un singur fir de execuție (`thread`)
- procesul fiu pornește de la zero în ce privește resursele utilizate și timpul de execuție precum și alți indicatori similari de gestionare a proceselor

Din momentul apelului, dacă acesta este cu succes, fiecare viitoare instrucțiune va fii executată atât de părinte cât și de copil. Diferențierea se face în funcție de ieșirea `fork(2)`: copilul primește valoarea 0 iar părintele `pid`-ul fiului. Astfel o construcție tipică C este

```
pid_t pid = fork();
if (pid < 0)
        return errno;
else if (pid == 0)
        /* child instructions */
else
        /* parent instructions */
```

Oricând în timpul execuției putem afla `pid`-ul procesului curent și pe cel al procesului părinte cu ajutorul funcțiilor `getpid(2)` și, respectiv, `getppid(2)`.

```
printf("Parent %d Me %d\n", getppid(), getpid());
```

Părintele își poate suspenda activitatea pentru a aștepta finalizarea execuției unui proces fiu cu ajutorul funcției `wait(2)`. `wait(2)` oferă la ieșire `pid`-ul fiului. **Atenție**, această funcție redă control părintelui când iese **oricare** dintre fii săi. Pentru cazuri complexe în care se dorește așteptarea unuia sau mai multor procese anume se pot folosii funcții avansate precum `waitpid(2)` sau `wait4(2)` care nu fac obiectul laboratorului.

Operația este utilă pentru a sincroniza și ordona instrucțiunile.

```
pid_t pid = fork();
if (pid < 0)
        return errno;
else if (pid == 0)
        /* child instructions */
        printf("First!");
else {
        /* parent instructions */
        wait(NULL);
        printf("Last!");
}
```

2 Executarea unui program existent

De multe ori datele sau rezultatele căutate pot fii obținute prin simpla execuție a unui program existent pe disk. Un mod de a ne folosi în procesul curent de alte programe este cu ajutorul funcției `execve(2)`.

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

Aceasta suprascrie complet procesul apelant cu un nou proces conform programului găsit la calea indicată în `path`. **Atenție**, calea trebuie să fie absolută! `/bin/pwd` nu `pwd`. Pentru a obține aceasta puteți folosi comanda `which(1)`

```
$ which pwd
/bin/pwd
$ which vi
/usr/bin/vi
```

Argumentele programului sunt puse în `argv` respectând convenția obișnuită din `C`: pe prima poziție (`argv[0]`) se află calea absolută către program urmată de argumente. Lista se încheie cu `null`. Variabilele de sistem din mediului de execuție sunt puse în ultimul argument `envp`. Aceasta este o listă de șiruri de caractere similară cu `argv` exceptând convenția primului element.

Datorită efectului distructiv asupra procesului curent, `execve(2)` este adesea folosit împreună cu `fork(2)` astfel încât procesul nou creat să fie cel suprascris.

```
pid_t pid = fork();
if (pid < 0)
        return errno;
else if (pid == 0)
        /* child instructions */
        char *argv[] = {"pwd", NULL};
        execve("/bin/pwd", argv, NULL);
        perror(NULL);
else
        /* parent instructions */
```

Pentru că suprascrierea procesului curent, `execve(2)` nu mai revine în programul inițial decât în cazul în care a apărut o eroare folosindu-se `errno` pentru a determina cauza. Cele mai des întâlnite erori sunt calea greșită sau lipsa lui `argv`.

3 Sarcini de laborator

1. Creați un proces nou folosind `fork(2)` și afișați fișierele din directorul curent cu ajutorul `execve(2)`. Din procesul inițial afișați `pid`-ul propriu și `pid`-ul copilului. De exemplu:

   ```
   $ ./forkls
   My PID=41875, Child PID=62668
   Makefile      collatz.c     forkls.c      so-lab-4.tex
   collatz       forkls        ncollatz.c
   Child 62668 finished
   ```

2. Ipoteza Collatz spune că plecând de la orice număr $n \in \mathbb{N}$ dacă aplicăm următorul algoritm

   $$n = \begin{cases} n/2 & \mod(n,2) = 0 \\ 3n+1 & \mod(n,2) \neq 0 \end{cases}$$

   seria va converge la 1. Implementați un program care folosește `fork(2)` și testează ipoteza generând secvența unui număr dat în procesul copil. Exemplu:

   ```
   $ ./collatz 24
   24: 24 12 6 3 10 5 16 8 4 2 1.
   Child 52923 finished
   ```

3. Implementați un program care să testeze ipoteza Collatz pentru mai multe numere date. Pornind de la un singur proces părinte, este creat câte un copil care se ocupă de un singur număr. Părintele va aștepta să termine execuția fiecare copil. Arătați că cerințele de sus sunt îndeplinite folosindu-vă de `getpid(2)` și `getppid(2)`. Exemplu:

```
$ ./ncollatz 9 16 25 36
Starting parent 6202
9: 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4
2 1.
36: 36 18 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5
16 8 4 2 1.
Done Parent 6202 Me 40018
Done Parent 6202 Me 30735
16: 16 8 4 2 1.
25: 25 76 38 19 58 29 88 44 22 11 34 17 52 26 13 40
20 10 5 16 8 4 2 1.
Done Parent 6202 Me 13388
Done Parent 6202 Me 98514
Done Parent 58543 Me 6202
```

Laboratorul 4

Procese

1 Crearea unui proces nou

În mediile de dezvoltare UNIX funcția sistem cu care se creează procese noi este funcția `fork(2)`. O dată invocată, funcția creează un proces nou (numit proces fiu sau proces copil) acesta fiind o copie a procesului apelant cu câteva excepții. Indicăm aici pe cele mai importante, ele pot diferi de la sistem de operare la sistem de operare

- procesul fiu are un ID unic (denumit și `pid`)
- procesul fiu are un părinte diferit
- procesul fiu are un singur fir de execuție (`thread`)
- procesul fiu pornește de la zero în ce privește resursele utilizate și timpul de execuție precum și alți indicatori similari de gestionare a proceselor

Din momentul apelului, dacă acesta este cu succes, fiecare viitoare instrucțiune va fii executată atât de părinte cât și de copil. Diferențierea se face în funcție de ieșirea `fork(2)`: copilul primește valoarea 0 iar părintele `pid`-ul fiului. Astfel o construcție tipică C este

```
pid_t pid = fork();
if (pid < 0)
        return errno;
else if (pid == 0)
        /* child instructions */
else
        /* parent instructions */
```

Oricând în timpul execuției putem afla `pid`-ul procesului curent și pe cel al procesului părinte cu ajutorul funcțiilor `getpid(2)` și, respectiv, `getppid(2)`.

```
printf("Parent %d Me %d\n", getppid(), getpid());
```

Părintele își poate suspenda activitatea pentru a aștepta finalizarea execuției unui proces fiu cu ajutorul funcției `wait(2)`. `wait(2)` oferă la ieșire `pid`-ul fiului. **Atenție**, această funcție redă control părintelui când iese **oricare** dintre fii săi. Pentru cazuri complexe în care se dorește așteptarea unuia sau mai multor procese anume se pot folosii funcții avansate precum `waitpid(2)` sau `wait4(2)` care nu fac obiectul laboratorului.

Operația este utilă pentru a sincroniza și ordona instrucțiunile.

```
pid_t pid = fork();
if (pid < 0)
        return errno;
else if (pid == 0)
        /* child instructions */
        printf("First!");
else {
        /* parent instructions */
        wait(NULL);
        printf("Last!");
}
```

2 Executarea unui program existent

De multe ori datele sau rezultatele căutate pot fii obținute prin simpla execuție a unui program existent pe disk. Un mod de a ne folosi în procesul curent de alte programe este cu ajutorul funcției `execve(2)`.

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

Aceasta suprascrie complet procesul apelant cu un nou proces conform programului găsit la calea indicată în `path`. **Atenție**, calea trebuie să fie absolută! `/bin/pwd` nu `pwd`. Pentru a obține aceasta puteți folosi comanda `which(1)`

```
$ which pwd
/bin/pwd
$ which vi
/usr/bin/vi
```

Argumentele programului sunt puse în `argv` respectând convenția obișnuită din `C`: pe prima poziție (`argv[0]`) se află calea absolută către program urmată de argumente. Lista se încheie cu `null`. Variabilele de sistem din mediului de execuție sunt puse în ultimul argument `envp`. Aceasta este o listă de șiruri de caractere similară cu `argv` exceptând convenția primului element.

Datorită efectului distructiv asupra procesului curent, `execve(2)` este adesea folosit împreună cu `fork(2)` astfel încât procesul nou creat să fie cel suprascris.

```
pid_t pid = fork();
if (pid < 0)
        return errno;
else if (pid == 0)
        /* child instructions */
        char *argv[] = {"pwd", NULL};
        execve("/bin/pwd", argv, NULL);
        perror(NULL);
else
        /* parent instructions */
```

Pentru că suprascrierea procesului curent, `execve(2)` nu mai revine în programul inițial decât în cazul în care a apărut o eroare folosindu-se `errno` pentru a determina cauza. Cele mai des întâlnite erori sunt calea greșită sau lipsa lui `argv`.

3 Sarcini de laborator

1. Creați un proces nou folosind `fork(2)` și afișați fișierele din directorul curent cu ajutorul `execve(2)`. Din procesul inițial afișați `pid`-ul propriu și `pid`-ul copilului. De exemplu:

   ```
   $ ./forkls
   My PID=41875, Child PID=62668
   Makefile      collatz.c     forkls.c      so-lab-4.tex
   collatz       forkls        ncollatz.c
   Child 62668 finished
   ```

2. Ipoteza Collatz spune că plecând de la orice număr $n \in \mathbb{N}$ dacă aplicăm următorul algoritm

   $$n = \begin{cases} n/2 & \mod(n,2) = 0 \\ 3n+1 & \mod(n,2) \neq 0 \end{cases}$$

   seria va converge la 1. Implementați un program care folosește `fork(2)` și testează ipoteza generând secvența unui număr dat în procesul copil. Exemplu:

   ```
   $ ./collatz 24
   24: 24 12 6 3 10 5 16 8 4 2 1.
   Child 52923 finished
   ```

3. Implementați un program care să testeze ipoteza Collatz pentru mai multe numere date. Pornind de la un singur proces părinte, este creat câte un copil care se ocupă de un singur număr. Părintele va aștepta să termine execuția fiecare copil. Arătați că cerințele de sus sunt îndeplinite folosindu-vă de `getpid(2)` și `getppid(2)`. Exemplu:

```
$ ./ncollatz 9 16 25 36
Starting parent 6202
9: 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4
2 1.
36: 36 18 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5
16 8 4 2 1.
Done Parent 6202 Me 40018
Done Parent 6202 Me 30735
16: 16 8 4 2 1.
25: 25 76 38 19 58 29 88 44 22 11 34 17 52 26 13 40
20 10 5 16 8 4 2 1.
Done Parent 6202 Me 13388
Done Parent 6202 Me 98514
Done Parent 58543 Me 6202
```

Laboratorul 5

Comunicare Inter-Proces

1 Memoria Partajată

În mediile de dezvoltare care respectă standardul POSIX (toate variantele UNIX și în mare parte Windows), obiectele de memorie partajată se creează cu ajutorul funcției `shm_open(3)`

```
int shm_open(const char *path, int flags, mode_t mode);
```

având o semantică aproape identică cu funcția de sistem `open(2)` (vezi Laboratorul 2) motiv pentru care `shm_open(3)` este deobicei doar un wraper peste aceasta. Argumentul `path` este de fapt numele obiectului și nu o cale în sistemul de fișiere. Un apel tipic arată astfel

```
char shm_name[] = "myshm";
int shm_fd;

shm_fd = shm_open(shm_name, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR);
if (shm_fd < 0) {
        perror(NULL);
        return errno;
}
```

unde obiectul `"myshm"`, care dacă nu există este creat (`O_CREAT`), este deschis pentru scriere și citire (`O_RDWR`) oferind drepturi asupra lui doar utilizatorului care l-a creat (`S_IRUSR | S_IWUSR`, vezi Laboratorul 2 și `chmod(2)`). Rezultatul este un descriptor `shm_fd` pe care îl putem folosi mai departe în orice funcție ce manipulează obiecte cu ajutorul descriptorilor precum toate funcțiile de sistem sau din biblioteca standard de C pentru fișiere.

O dată creat primul pas este să îi definim dimensiunea cu ajutorul funcției de sistem `ftruncate(2)`

```
size_t shm_size = 1000;

if (ftruncate(shm_fd, shm_size) == -1) {
        perror(NULL);
        shm_unlink(shm_name);
        return errno;
}
```

Aceasta scurtează sau mărește obiectul asociat descriptorului dat conform noii dimensiuni primite în al doilea argument. În exemplul nostru mărește obiectul `shm_fd` de la 0 bytes la 1000.

Funcția `shm_unlink(3)` șterge obiectele create cu funcția `shm_open(3)` primind numele obiectului ca parametru. Aceasta este din nou o extindere firească de la funcția de sistem `unlink(2)` folosită în mod normal pentru a șterege fișiere de pe disk. Un apel tipic poate fii văzut în exemplul `ftruncate(2)` de mai devreme.

Memoria partajată se încarcă în spațiul procesului cu ajutorul funcției de sistem `mmap(2)`.

```
void * mmap(void *addr, size_t len, int prot, int flags,
            int fd, off_t offset);
```

Parametrii sunt

- `addr` – adresa la care să fie încărcată în proces (de obicei aici folosim 0 pentru a lăsa kernelul să decidă unde încarcă)
- `len` – dimensiunea memoriei încărcate
- `prot` – drepturile de acces (`PROT_READ` sau `PROT_WRITE` de obicei)
- `flags` – tipul de memorie (de obicei `MAP_SHARED` astfel încât modificările făcute de către proces să fie vizibile și în celelalte)
- `fd` – descriptorul obiectului de memorie
- `offset` – locul în obiectul de memorie partajată de la care să fie încărcat în spațiul procesului

iar, când se execută cu succes, rezultatul este un pointer către adresa din spațiul procesului la care a fost încărcat obiectul. Altfel, valoarea `MAP_FAILED` este întoarsă și `errno` este setat corespunzător.

Apelurile cele mai des întâlnite sunt de tipul

```
shm_ptr = mmap(0, shm_size, PROT_READ, MAP_SHARED, shm_fd, 0);
if (shm_ptr == MAP_FAILED) {
        perror(NULL);
        shm_unlink(shm_name);
        return errno;
}
```

unde `shm_ptr` va indica către **toată** zona de memorie (`shm_size`) aferentă descriptorului (`shm_fd`) care va fii doar citită (`PROT_READ`) și împărțită cu restul proceselor (`MAP_SHARED`), sau de tipul

```
shm_ptr = mmap(0, 100, PROT_WRITE, MAP_SHARED, shm_fd, 500);
if (shm_ptr == MAP_FAILED) {
        perror(NULL);
        shm_unlink(shm_name);
        return errno;
}
```

unde `shm_ptr` va indica către o **parte** de **100** bytes care începe de la byte-ul **500** din zona de memorie aferentă descriptorului (`shm_fd`) ce va fii doar scrisă (`PROT_WRITE`) și împărțită cu restul proceselor (`MAP_SHARED`).

ATENȚIE: în Linux dimensiunea trebuie să fie multiplu de pagini: `PAGE_SIZE`. Se poate obține și cu `getpagesize(2)`.

Când nu mai este nevoie de zona de memorie încărcată, se folosește funcția `munmap(2)`

```
int munmap(void *addr, size_t len);
```

care primește pointer-ul către zona încărcată în spațiul procesului și dimensiunea ca argumente. Pentru exemplele anterioare am folosi

```
munmap(shm_ptr, shm_size);
```

pentru când a fost încărcată **toată** zona, și

```
munmap(shm_ptr, 100)
```

pentru când a fost încărcată o **parte**.

2 Sarcini de laborator

1. Ipoteza Collatz spune că plecând de la orice număr $n \in \mathbb{N}$ dacă aplicăm următorul algoritm

$$n = \begin{cases} n/2 & \mod(n,2) = 0 \\ 3n+1 & \mod(n,2) \neq 0 \end{cases}$$

   Implementați un program care să testeze ipoteza Collatz pentru mai multe numere date folosind memorie partajată.

   Pornind de la un singur proces părinte, este creat câte un copil care se ocupă de un singur număr și scrie seria undeva în memoria partajată. Părintele va crea obiectul de memorie partajată folosind `shm_open(3)` și `ftruncate(2)` și pe urmă va încărca în memorie întreg spațiul pentru citirea rezultatelor cu `mmap(2)`.

O convenție trebuie stabilită între părinte și fii astfel încât fiecare copil să aibă acces exclusiv la o parte din memoria partajată unde își va scrie datele (ex. împărțim memoria în mod egal pentru fiecare copil). Astfel, fiecare copil va încărca doar zona dedicată lui pentru scriere folosind dimensiunea cuvenită și un deplasament nenul în `mmap(2)`. Părintele va aștepta să termine execuția fiecare copil după care va scrie pe ecran rezultatele obținute de fii săi.

Arătați că cerințele de sus sunt îndeplinite folosindu-vă de `getpid(2)` și `getppid(2)`. Exemplu:

```
$ ./shmcollatz 9 16 25 36
Starting parent 75383
Done Parent 75383 Me 59702
Done Parent 75383 Me 3281
Done Parent 75383 Me 33946
Done Parent 75383 Me 85263
9: 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4
2 1.
16: 16 8 4 2 1.
25: 25 76 38 19 58 29 88 44 22 11 34 17 52 26 13 40
20 10 5 16 8 4 2 1.
36: 36 18 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5
16 8 4 2 1.
Done Parent 96028 Me 75383
```

2. În programul anterior folosiți `shm_unlink(3)` și `munmap(2)` pentru a elibera resursele folosite.

Laboratorul 6

Fire de execuție

1 Crearea firelor de execuție

Laboratoarele anterioare au discutat despre crearea unor procese noi cu ajutorul funcțiilor de tip `fork(2)` sau `execve(2)`. Procesele noi create erau o copie fidelă a celui inițial, dar resuresele erau complet separate fiind necesare diferite mecanisme de comunicație inter-proces pentru a colabora.

Pentru ușurarea comunicării și sporirea performanței se pot folosi fire de execuție separate ale aceluiași proces. Acestea au avantajul că împart toate resursele și orice modificare făcută în spațiul procesului de un fir este instantaneu vizibilă tuturor celorlalte fără a apela la un mecanism exterior.

Dezavantajele apar o dată cu nevoia de a scrie și/sau citi concomitent aceiași zonă de memorie. În acest caz, concurența și drepturile de acces asupra resursei trebuie dictate de terți îngreunând astfel procesul de execuție și de multe ori introducând defecte subtile în program.

Firele de execuție, denumite thread în literatura de specialitate, sunt implementate în mediile POSIX prin variabile de tip `pthread_t`. Pentru a creea un thread nou se folosește funcția `pthread_create(3)`

```
int
pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine)(void *), void *arg);
```

care inițializează `thread` cu noul fir de execuție lansat prin apelarea funcției `start_routine` cu argumentele oferite de `arg`. **Atenție**, diferit de procesele create cu `fork(2)`, un thread nou pornește execuția de la o funcție dată.

La inițializare, se pot particulariza anumite detalii privind noul thread (ex. dimensiunea stivei, chestiuni de securitate etc.). Pe parcursul laboratorului vom folosi atributele implicit setate de sistemul de operare, semnalând aceasta prin folosirea valorii `NULL` pentru `attr`.

Un apel tipic pentru lansarea unui nou fir de execuție este

```
pthread_t thr;
if (pthread_create(&thr, NULL, hello, "world!")) {
        perror(NULL);
        return errno;
}
```

unde funcția `hello` trebuie să respecte prototipul `start_routine` de mai devreme

```
void *
hello(void *v)
{
        char *who = (char *)v;
        printf("Hello, %s!", who);
        return NULL;
}
```

Pentru a aștepta finalizarea execuției unui thread se folosește `pthread_join(3)`

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

care, diferit de `wait(2)`, așteaptă explicit firul de execuție din variabila `thread`. Dacă `value_ptr` nu este `NULL`, atunci `pthread_join` va pune la adresa indicată rezultatul funcției `start_routine`. Un apel tipic este

```
if (pthread_join(thr, &result)) {
        perror(NULL);
        return errno;
}
```

În mediile POSIX funcționalitatea thread se găsește într-o bibliotecă separată numită `libpthread`. Astfel la compilare este nevoie să specificăm explicit această legătură

```
$ cc hello.c -o hello -pthread
```

2 Sarcini de laborator

1. Scrieți un program care primește un șir de caractere la intrare ale cărui caractere le copiază în în ordine inversă și le salvează într-un șir separat. Operație de inversare va avea loc într-un thread separat. Rezultatul va fi obținut cu ajutorul funcției `pthread_join`. Exemplu

   ```
   $ ./strrev hello
   olleh
   ```

2. Scrieți un program care să calculeze produsul a două matrice date (de dimensiuni compatibile) unde fiecare element al matricei rezultate este calculat de către un thread distinct.

   Reamintim că date $\boldsymbol{A} \in \mathbb{R}^{m \times p}$ și $\boldsymbol{B} \in \mathbb{R}^{p \times n}$, elementul c_{ij} al matricei $\boldsymbol{C} \in \mathbb{R}^{m \times n}$ este calculat cu ajutorul formulei

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}.$$

Laboratorul 7

Sincronizare

1 Excluziune mutuală

Adesea în programele cu mai multe fire de execuție și procese avem nevoie ca o singură entitate să execute un număr de instrucțiuni la un moment de timp dat. Această zonă în care are voie un singur proces sau thread se numește zonă critică (critical section).

La laborator vom folosi exclusiv pentru paralelism fire de execuție, dar conceptele folosite se pot aplica la fel de bine pentru situația în care avem de-a face cu mai mult procese.

Obiectul cel mai des folosit pentru asigurarea accesului exclusiv într-o zonă critică este mutex-ul (prescurtat de la **mut**ual **ex**clusive). În bibliotecile de sistem ce implementează standardul POSIX, un obiect de tip mutex este creat cu ajutorul funcției `pthread_mutex_init(3)`

```
int pthread_mutex_init(pthread_mutex_t *mutex,
 const pthread_mutexattr_t *attr);
```

care inițializează un `mutex` cu atributele `attr` (la laborator se va folosi `NULL` pentru a obține atributele implicite similar cum am procedat pentru `pthread`). De obicei variabila mutex se pune fie în memoria globală, fie într-o structură accesibilă tuturor firelor de execuție după care se inițializează în `main()` sau undeva înainte de a fii folosit.

```
pthread_mutex_t mtx;
int main()
{
        /* ... */
        if (pthread_mutex_init(&mtx, NULL)) {
                perror(NULL);
                return errno;
        }
        /* ... */
}
```

Un mutex se poate afla în două stări: fie închis (locked), fie deschis (unlocked). Când mutexul este închis înseamnă că un thread deține dreptul exclusiv de execuție asupra zonei critice până ce decide să renunțe la acest drept.

Pentru a obține un mutex (i.e. pentru a-l închide) se folosește funcția `pthread_mutex_lock(3)` iar pentru a-l elibera se folosește `pthread_mutex_unlock(3)`. Ele sunt folosite tipic în tandem, demarcând zona critică

```
pthread_mutex_lock(&mtx);
count++;        /* critical section */
pthread_mutex_unlock(&mtx);
```

Atenție, funcția `pthread_mutex_lock(3)` nu se termină de executat până nu obține mutexul, blocând astfel firul de execuție ce a apelat-o!

La sfârșit, când nu mai avem nevoie de obiectul de tip mutex, eliberăm resursele ocupate cu `pthread_mutex_destroy(3)`

```
pthread_mutex_destroy(&mtx);
```

2 Semafoare

Semafoarele sunt similare obiectelor de tip mutex, dar pot face față unor scenarii mai sofisticate de sincronizare.

În esență, un semafor este o variabilă S inițializată cu o valoare întreagă care este manipulată exclusiv cu ajutorul a două funcții: `wait` și `post` (sau `signal`). În principiu, ele scad, respectiv cresc, cu o unitate valoarea lui S.

Condiția principală ca obiectele de tip semafor să funcționeze corect este ca funcțiile de mai sus să se execute fără a fii întrerupte (să se execute atomic). Astfel putem observa că un obiect de tip mutex este defapt un caz particular de semafor în care $S = 1$.

În mediile POSIX semafoarele sunt inițializate cu `sem_init(3)`

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

care setează valoarea inițială a semaforului `sem` cu $S =$ `value`. Variabila `pshared` este folosită pentru a semnala dacă vrem să folosim semaforul în cadrul mai multor procese. Nefiind cazul în cadrul laboratorului, vom folosi tot timpul `pshared` $= 0$. Apelul tipic este similar cu cel în cazul obiectelor mutex

```
sem_t sem;
int main()
{
        /* ... */
        if (sem_init(&sem, 0, S)) {
                perror(NULL);
                return errno;
        }
        /* ... */
}
```

unde S este valoarea inițială aleasă.

Funcția `sem_wait(3)` scade valoarea lui S cu o unitate și poate fii executată cu siguranță în medii de lucru paralele.

```
if (sem_wait(&sem)) {
        perror(NULL);
        return errno;
}
```

Atenție, dacă $S = 0$ atunci funcția așteaptă ca valoarea să crească înainte de a o scădea blocând astfel firul apelant!

Similar, pentru a crește valoarea lui S este folosită funcția `sem_post(3)`

```
if (sem_post(&sem)) {
        perror(NULL);
        return errno;
}
```

care, după incrementare $S = S + 1$, verifică dacă sunt thread-uri blocate de semafor și eliberează thread-ul care așteaptă de cel mai mult timp în coadă. Acesta va relua execuția din punctul în care a apelat `sem_wait(3)`.

Când nu mai este folosit, semaforul este eliberat cu ajutorul funcției `sem_destroy(3)`

```
sem_destroy(&sem);
```

3 Sarcini de laborator

1. Scrieți un program care gestionează accesul la un număr finit de resurse. Mai multe fire de execuție pot cere concomitent o parte din resurse pe care le vor da înapoi o dată ce nu le mai sunt utile. Fie numărul maxim de resurse dat

   ```
   #define MAX_RESOURCES 5
   int available_resources = MAX_RESOURCES;
   ```

   Când un thread dorește să oțină un număr de resurse acesta apelează `decrease_count`.

   ```
   int decrease_count(int count)
   {
           if (available_resources < count)
                   return -1;
           else
                   available_resources -= count;
           return 0;
   }
   ```

   iar când resursele nu-i mai sunt necesare apelează `increase_count`

```
int increase_count(int count)
{
        available_resources += count;
        return 0;
}
```

Funcțiile de mai sus prezintă mai multe defecte într-un mediu de execuție paralel printre care și un **race condition**. Modificați funcțiile și rezolvați race condition-ul folosind obiecte de tip mutex. Arătați că modificările dumneavoastră sunt corecte cu ajutorul unui program care pornește mai multe thread-uri ce consumă un număr diferit de resurse fiecare. De exemplu:

```
$ ./count
MAX_RESOURCES=5
Got 2 resources 3 remaining
Released 2 resources 5 remaining
Got 2 resources 3 remaining
Released 2 resources 5 remaining
Got 1 resources 4 remaining
Released 1 resources 5 remaining
Got 3 resources 2 remaining
Got 2 resources 0 remaining
Released 3 resources 3 remaining
Released 2 resources 5 remaining
```

2. Scrieți un program care să sincronizeze execuția a N fire de execuție construind un obiect de tip barieră. Bariera va fii inițializată folosind `init(N)` și fiecare thread va apela `barrier_point()` când va ajunge în dreptul barierei. Când funcția este apelată a N-a oară, aceasta pornește execuția tuturor firelor în așteptare.
   Verificați rezultatele dumneavoastră cu ajutorul unui program care pornește mai multe thread-uri ce se folosesc de barieră pentru a-și sincroniza execuția.

   Funcția executată de fiecare fir poate avea următoarea formă

```
void * tfun(void *v)
{
        int *tid = (int *)v;

        printf("%d reached the barrier\n", *tid);
        barrier_point();
        printf("%d passed the barrier\n", *tid);

        free(tid);
        return NULL;
}
```

unde `tid` este numărul threadului pornit. Astfel, o instanță cu 5 fire de execuție ar afișa

```
$ ./barrier
NTHRS=5
0 reached the barrier
1 reached the barrier
2 reached the barrier
3 reached the barrier
4 reached the barrier
4 passed the barrier
0 passed the barrier
1 passed the barrier
3 passed the barrier
2 passed the barrier
```

Indiciu, pentru a implementa obiectul de tip barieră folosiți un mutex pentru contorizarea firelor ajunse la barieră și un semafor pentru a aștepta la barieră.

Laboratorul 1

Linia de comandă și execuția

1 Familiarizarea cu terminalul

În Tabelul 1 găsiți câteva comenzi utile navigării în terminal. Pentru fiecare citiți bine prima parte din manual. De exemplu pentru comanda `cp(1)` manualul începe cu

```
CP(1)                General Commands Manual                CP(1)

NAME
     cp - copy files

SYNOPSIS
     cp [-fipv] [-R [-H | -L | -P]] source target
     cp [-fipv] [-R [-H | -L | -P]] source ... directory

DESCRIPTION
     In the first synopsis form, the cp utility copies the
     contents of the source file to the target file.
```

Prima linie ne spune că ne aflăm în secțiunea `(1)` numită `General Commands Manual`. Ce urmează după numele comenzii în paranteză specifică secțiunea. Mai departe avem rezumatul comenzii și tipul de parametrii pe care-i acceptă. După introducere urmează descrierea pe larg. Pentru a vedea o comandă dintr-o secțiune anume folosiți

```
$ man 1 write
$ man 2 write
```

prima comandă descrie `write(1)` iar a doua syscall-ul `write(2)`.

Câteva operații și simboluri folosite în linia de comandă sunt descrise în Tabelul 2.

Urmează o sesiune exemplu în terminal unde folosim câteva din comenzile și operațiile descrise mai sus.

Comandă	Descriere
`man command`	manualul de utilizare
`pwd`	directorul curent
`ls`	conținutul directorului curent
`cp source target`	copiere fișiere
`mv source target`	mutare fișiere
`rm item`	ștergere fișiere
`mkdir dir`	creeare director
`rmdir dir`	ștergere director gol
`echo str`	repetare string
`cd path`	schimbă directorul curent

Tabela 1: Comenzi uzuale

Simbol	Descriere
`.`	directorul curent
`..`	directorul părinte
	acasă (`/home/souser`)
`cmd > file`	redirecționare ieșire către fișier
`cmd1 | cmd2`	pipe: legătură ieșire-intrare
`^`	tasta `ctrl`
`^w`	tastat concomitent `ctrl+w`

Tabela 2: Simboluri și operații în terminal

```
$ pwd
/home/souser
$ touch foo
$ ls
foo
$ cp foo bar
$ ls
bar foo
$ mv bar baz
$ ls
baz foo
$ rm baz
$ ls
foo
$ mkdir test
$ cd test/
$ pwd
/home/souser/test
$ cd ..
$ rmdir test
$ echo hello
```

```
hello
$ echo hello > hello.txt
$ cat hello.txt
hello
```

2 Analizarea execuției unui binar

Pentru a observa cum încarcă sistemul de operare un executabil de pe disk, în cele ce urmează vom crea un executabil simplu de tip helloworld.

```
#include <stdio.h>

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

Acest program se poate scrie cu ajutorul oricărui editor. În linia de comandă se pot folosi `nano(1)`, `vi(1)`, `emacs(1)` sau oricare alt editor. Subiectul acesta nu face obiectul laboratorului.

2.1 Funcții sistem

Mai deaprte, analizăm funcțiile de sistem (syscall) folosite pentru a duce executa binarul

```
$ gcc helloworld.c -o hello
$ ./hello
Hello, World!
$ ktrace hello
Hello, World!
$ kdump
```

Comanda `ktrace(1)`, scurt de la kernel trace, ne ajută să vedem de ce funcții de sistem are nevoie `hello` pentru a fii executat. Echivalentul în Linux este `strace(1)`. `kdump(1)` ne ajută să vedem fișierul binar creat de `ktrace(1)`. Pornirea execuției are loc prin apelul la `execve(2)`

```
46707 ktrace   CALL  execve(0x7f7ffffd5f08,0x7f7ffffd5da0,0x7f7ffffd5db0)
46707 ktrace   NAMI  "./hello"
46707 ktrace   ARGS
       [0] = "./hello"
41281 hello    NAMI  "/usr/libexec/ld.so"
41281 hello    RET   execve 0
```

valorile hexazecimale sunt adrese în memorie corespunzătoare argumentelor apelului de sistem. Pentru a le descifra, folosiți manualul (`$ man 2 execve`).

```
EXECVE(2)                System Calls Manual                EXECVE(2)

NAME
     execve - execute a file

SYNOPSIS
     #include <unistd.h>

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

Părțile direct influențate de programul nostru sunt

```
41281 hello     CALL  write(1,0x7bd390d0000,0xe)
41281 hello     GIO   fd 1 wrote 14 bytes
      "Hello, World!
      "
41281 hello     RET   write 14/0xe
```

unde mesajul este afișat pe ecran cu ajutorul syscall-ului `write(2)`. Conform `$ man 2 execve`

```
WRITE(2)                 System Calls Manual                WRITE(2)

NAME
     write, writev, pwrite, pwritev - write output

SYNOPSIS
     #include <unistd.h>

     ssize_t
     write(int d, const void *buf, size_t nbytes);
     [...]
DESCRIPTION
     write() attempts to write nbytes of data to the object
     referenced by the descriptor d from the buffer pointed
     to by buf.
```

Descriptorii sunt folosiți pentru a indica către anumite fișiere din sistem. Înloc de a folosi `/dir/subdir/file.txt` putem scurta cu un număr care știm că atunci când apare se referă la acest fișier (structură de tip cheie–valoare). Primii trei descriptori sunt rezervați. Detalii în Tabelul 3.

2.2 Biblioteci

Funcția `printf(3)` folosită în `helloworld.c` este implementată în biblioteca standard de C numită `libc`. Cu ajutorul utilitarului `ldd(1)` putem vedea de ce biblioteci are nevoie `hello` pentru a fii executat

Descriptor	Fișier	Folosit de
0	`stdin`	`scanf(3)`
1	`stdout`	`printf(3)`
2	`stderr`	`perror(3)`

Tabela 3: Descriptori rezervați

```
$ ldd ./hello
./hello:
Start            End              Type Open Ref GrpRef Name
9e7c6e00000      9e7c7002000      exe  1    0   0      ./hello
9ea8816b000      9ea8844a000      rlib 0    1   0      /usr/lib/libc.so.90.0
9ea60800000      9ea60800000      rtld 0    1   0      /usr/libexec/ld.so
```

Primele coloane indică unde în memorie poate fii găsită fiecare bibliotecă. Putem vedea că pe lângă `libc` mai este necesar și `ld.so`. Aceasta din urmă este o bibliotecă specifică cu care sistemul de operare caută, găsește și încarcă în memorie bibliotecile utilizate de executabil (în cazul nostru doar `libc`).

2.3 Simboluri

Pentru a vedea de ce simboluri are nevoie `hello`, putem folosi `nm(1)`

```
$ nm ./hello
00200e50 a _DYNAMIC
00200fb0 a _GLOBAL_OFFSET_TABLE_
         W _Jv_RegisterClasses
00201000 A __bss_start
00201000 A __data_start
00201000 B __dso_handle
00000550 T __fini
00200e40 D __guard_local
00000350 T __init
00000420 W __register_frame_info
000003c0 T __start
         U _csu_finish
00201000 A _edata
00201058 A _end
000003c0 T _start
         U atexit
         U exit
00000000 F helloworld.c
00000528 T main
         U puts
```

Prima coloană arată adresa la care se găsește fiecare simbol. A doua indică tipul simbolului (`U` este prescurtarea de la `Undefined`) iar ultima numele. Executa-

Comandă	Descriere
`b symbol`	oprirea execuției la simbol
`p var`	tipărește valoarea variabilei
`n`	următoarea instrucțiune
`c`	continuarea execuției
`q`	ieșire

Tabela 4: Comenzi `gdb(1)`

bilul nostru folosește explicit doar `printf(3)` din biblioteca standard care în exemplul de sus este implementat cu ajutorul lui `puts(3)` iar adresa la care se v-a găsi această funcție va fii stabilită când se va încărca în memorie biblioteca C în timpul execuției (vezi secțiunea precedentă). Un alt simbol cunoscut este `main` și `helloworld.c` (fișierul sursă).

2.4 Instrumentare

Un prim utilitar folosit la investigarea și instrumentarea executabilelor este `gdb(1)`. Întâi vom recompila cu simboluri de debug

```
$ gcc -g -O0 helloworld.c -o hello
```

Opțiunile în plus sunt `-g` care adaugă efectiv simbolurile (numărul liniei în fișierul sursă, instrucțiunea C etc.) și `-O0` pentru a elimina optimizările compilatorului ce ar putea rezulta în eliminarea anumitor variabile sau instrucțiuni C pierzându-se astfel corespondența cu fișierul sursă.

În Tabelul 4 sunt puse câteva instrucțiuni uzuale folosite pentru a depana cu `gdb`. încheiem cu o sesiune de depanare unde se observă faptul că breakpoint-ul se pune de fapt la o adresă în memorie deși argumentul este un simbol și liniile din fișierul sursă sunt afișate corespunzător.

```
$ gdb ./hello
(gdb) b main
Breakpoint 1 at 0x530: file helloworld.c, line 5.
(gdb) r
Starting program: /home/souser/hello
Breakpoint 1 at 0xbac61500530: file helloworld.c, line 5.

Breakpoint 1, main () at helloworld.c:5
5               printf("Hello, World!\n");
(gdb) n
Hello, World!
6               return 0;
(gdb) n
7       }
(gdb) n
0x00000bac61500414 in _start () from /home/souser/hello
```

```
(gdb) n
Single stepping until exit from function _start,
which has no line number information.

Program exited normally.
(gdb) q
```

Laboratorul 2

Funcții sistem

1 Utilizarea funcțiilor sistem

Reamintim faptul că funcțiile de sistem (syscalls) sunt definite în secțiunea 2 a manualului sistemului de operare. Aprofundați definiția și utilizarea fiecărei funcții folosite în acest material prin apeluri de tipul

```
$ man 2 <syscall>
```

Funcțiile cele mai des întâlnite pentru manipularea fișierelor sunt `read(2)`, `write(2)`, `stat(2)`, `open(2)`, și `close(2)`.

1.1 Citire și scriere

Am văzut în Laboratorul 1 cum se comportă `write(2)`. Similar, `read(2)` citește dintr-un descriptor `d` în buferul `buf` un număr dat de `nbytes`.

```
ssize_t read(int d, void *buf, size_t nbytes);
```

Când este executată cu succes, ieșirea funcției este numărul de bytes citiți.

1.2 Accesarea fișierelor

Pentru a obține un descriptor asociat unui fișier trebuie folosită funcția `open(2)` care deschide fișierul găsit în `path` pentru scriere și/sau citire.

```
int open(const char *path, int flags, ...);
```

Ieșirea funcției este descriptorul asociat. Modul în care va fi manipulat fișierul este dat de argumentul `flags` similar funcției standard C `fopen(3)`.

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.

Dacă fișierul cerut nu există în sistem, se poate cere creearea lui prin adăugarea flagului `O_CREAT` la cele de scriere sau citire. În acest caz, trebuie specificate și drepturile de acces la fișier în al treilea argument.

```
#define S_IRWXU 0000700     /* RWX mask for owner */
#define S_IRUSR 0000400     /* R for owner */
#define S_IWUSR 0000200     /* W for owner */
#define S_IXUSR 0000100     /* X for owner */
```

Vezi manualul `open(2)` și tabelul din `chmod(2)` pentru mai multe detalii.

Orice fișier deschis cu `open(2)` trebuie închis când nu mai este folosit cu `close(2)`.

1.3 Informații despre fișiere

Pentru a afla detlaii despre obiectele manipulate precum dimensiunea ocupată pe disk, permisiunile de acces, data la care a fost creat și modificat ultima dată, se folosește funcția `stat(2)`.

```
int stat(const char *path, struct stat *sb);
```

În câmpurile structurii de date `stat` vor fii populate informațiile de mai sus împreună cu alte detalii.

```
struct stat {
 dev_t     st_dev;     /* inode's device */
 ino_t     st_ino;     /* inode's number */
 mode_t    st_mode;    /* inode protection mode */
 nlink_t   st_nlink;   /* number of hard links */
 uid_t     st_uid;     /* user ID of the file's owner */
 gid_t     st_gid;     /* group ID of the file's group */
 dev_t     st_rdev;    /* device type */
 struct timespec st_atim;  /* time of last access */
 struct timespec st_mtim;  /* last data modification */
 struct timespec st_ctim;  /* last file status change */
 off_t     st_size;    /* file size, in bytes */
 blkcnt_t  st_blocks;  /* blocks allocated for file */
 blksize_t st_blksize;/* optimal blocksize for I/O */
 u_int32_t st_flags;   /* user defined flags for file */
 u_int32_t st_gen;     /* file generation number */
 };
```

Următorul fragment de program afișează dimensiunea fișierului `foo`.

```
#include <sys/stat.h>
...
struct stat sb;
if (stat("foo", &sb)) {
        perror("foo");
```

errno	Valoare	Descriere
1	EPERM	operația nu este permisă
2	ENOENT	fișier sau director inexistent
5	EIO	eroare de comunicare intrare/ieșire (cu un dispozitiv)
9	EBADF	descriptor inexistent
12	ENOMEM	memorie insuficientă
13	EACESS	nu sunt permisiuni suficiente de acces
14	EFAULT	adresă invalidă
22	EINVAL	argument invalid

Tabela 1: Coduri de eroare uzuale

```
        return errno;
}
printf("Foo takes %jd bytes on disk\n", sb.st_size);
```

2 Tratarea erorilor

În manualele de utilizare există o secțiune importantă numită `RETURN VALUES`. Adesea valoarea la ieșirea cu succes este pozitivă, iar când apelul întâmpină o problemă utilizatorul este semnalat prin valoarea -1. În acest caz mai multe detalii se pot găsi în variabila globală `ernno`. Codul de eroare indicat are asociat un mesaj de eroare ce poate fii ușor afișat pe ecran cu ajutorul funcției `perror(3)`.

Funcția `read(2)` spune următoarele în documentație

> **RETURN VALUES**
> If successful, the number of bytes actually read is returned. Upon reading end-of-file, zero is returned. Otherwise, a -1 is returned and the global variable errno is set to indicate the error.

deci un apel corect al funcției arată astfel

```
nread = read(fd, buf, bufsz);
if (nread < 0) {
        perror("read buf");
        return errno;
}
```

în anumite cazuri se poate face un caz special și pentru `nread == 0` semnalând că am ajuns la sfârșitul fișierului.

În Tabelul 1 puteți găsi câteva din cele mai frecvente erori semnalate de errno. O listă completă cu valorile posibile și semnificația lor se găsește în manual `errno(2)`.

Toate apelurile de funcții trebuie verificate corespunzător pentru toate ieșirile posibile, fie cu succes, fie fără!

3 Sarcini de laborator

1. Rescrieți programul HelloWorld de data trecută folosind numai funcții sistem.

2. Scrieți un program `mycp` care să primească la intrare în primul argument un fișier sursă pe care să-l copieze într-un alt fișier cu numele primit în al doilea argument. Exemplu apel `./mycp foo bar`.

Laboratorul 3

Implementare funcții sistem

1 Compilare kernel

Kernelul OpenBSD se găsește în `/bsd` iar codul sursă din care este compilat în `/sys`. Pentru a compila un kernel nou trebuie executate următoarele comenzi:

```
# cd /sys/arch/$(machine)/compile/GENERIC.MP
# make obj
# make config
# make
```

Înainte de a instala un kernel nou e bine să faceți o copie a originalului pentru a putea reveni în cazul în care noul kernel are un defect la pornirea sistemului de operare

```
# cp /bsd /bsd.1
```

după acest pas de siguranță, executați

```
# make install
```

și noul kernel va fii încărcat implicit la repornirea calculatorului

```
# reboot
```

2 Adăugarea unei noi funcții sistem

În OpenBSD funcțiile de sistem sunt definite în `/sys/kern/syscalls.master`. Din acest fișier se vor genera fișiere C care definesc structurile de date, varibilele și funcțiile necesare.

De exemplu pentru funcțiile cunoscute `read(2)` și `write(2)` veți găsi în acest fișier următoarele intrări

```
3   STD   { ssize_t sys_read(int fd, void *buf, size_t nbyte); }
4   STD   { ssize_t sys_write(int fd, const void *buf, \\
                              size_t nbyte); }
```

Primul câmp reprezintă numărul de identificare a funcției de sistem, al doilea tipul (în general noi vom folosi tot timpul funcții de sistem standard `STD`) iar ultimul câmp este definiția C a funcției prefixată cu `sys_`.

2.1 Declararea

Adăugarea unei noi intrări se face la sfârșitul fișierului `/sys/kern/syscalls.master`. ID-ul pentru funcția noastră va fi următoarea valoare după cea a ultimei funcții existente. De exemplu dacă ultimul ID este 330, noi vom folosi 331 pentru noua intrare.

```
331   STD   { int sys_khello(const char *msg); }
```

După modificarea fișierului `/sys/kern/syscalls.master` regenerați fișierele C aferente prin comanda

```
# cd /sys/kern && make syscalls
```

Fișierele generate sunt în directorul `/sys/kern` și `/sys/sys`. Modificările principale sunt

- `/sys/kern/syscalls.c` – adăugarea denumirii funcției în tabela `syscallnames`
- `/sys/sys/syscallargs.h` – definiția structurii ce va ține argumentele

  ```
  struct sys_khello_args {
         syscallarg(const char *) msg;
  };
  ```

- `/sys/sys/syscall.h` – definirea noului ID 331

  ```
  #define SYS_khello 331
  ```

  Declarația funcției are loc tot în `/sys/sys/syscallargs.h`

```
int sys_khello(struct proc *p, void *v, register_t *retval);
```

iar argumentele reale (din perspectiva kernelului) sunt

- `struct proc *p` – procesul care apelează
- `void *v` – pointer către structura `sys_khello_args`
- `register_t *retval` – pointer către rezultatul (ieșirea) funcției

Funcție	Apel	Descriere
`copyin(9)`	`copyin(ubuf, kbuf, len)`	copiază buffer user → kernel
`copyout(9)`	`copyout(kbuf, ubuf, len)`	copiază buffer kernel → user
`kcopy(9)`	`kcopy(srckbuf, dstkbuf, len)`	copiază buffer kernel → kernel
`copyinstr(9)`	`copyinstr(ubuf, kbuf, len, &done)`	copiază string user → kernel
`copyoutstr(9)`	`copyoutstr(kbuf, ubuf, len, &done)`	copiază string kernel → user
`copystr(9)`	`copystr(kbuf, kbuf, len, &done)`	copiază string kernel → kernel

Tabela 1: Funcții de copiere pentru kernel

2.2 Definirea

Funcția de sistem se definește de regulă în `/sys/kern/sys_generic.c`.

```
/*
 * Hello system call
 */
int
sys_khello(struct proc *p, void *v, register_t *retval)
{
        return 0;
}
```

Atenție, această funcție întoarce un `int` care conține un cod de eroare de tipul errno folosit mai departe de kernel. Valoarea pe care o întoarce în userland este diferită și trebuie pusă în argumentul `retval`.

Următorul pas este să citim argumentele de la intrare de la adresa indicată de `v`. Pentru asta trebuie să folosim structura `sys_khello_args` definită mai devreme.

```
        struct sys_khello_args *uap = v;
```

Conținutul structurii este comentat pentru ușurința programatorului, dar poate fi omis în funcție de gust.

Pentru a citi un argument se folosește macroul `SCARG`

```
#if _BYTE_ORDER == _BIG_ENDIAN
#define SCARG(p, k) ((p)->k.be.datum)
#elif _BYTE_ORDER == _LITTLE_ENDIAN
#define SCARG(p, k) ((p)->k.le.datum)
#else
#error "what byte order is this machine?"
#endif
```

care se ocupă cu încărcarea din registru care conține adresa la care se află structura cu argumentele trimise de utilizator (vezi Cursul 2). De exemplu, pentru a obține argumentul `msg` al noului nostru syscall `khello` folosim `SCARG(uap, msg)`.

Când avem de a face cu buffere primite din userland trebuie să le mutăm din spațiul de adresare specific procesului `p` în spațiul de adresare al kernelului. Pentru asta se pun la dispoziție seria de funcții din Tabelul 1. Mecanismul trebuie folosit în cadrul funcției de sistem `khello` pentru a copia primii 100 bytes din mesajul `msg` în spațiul kernelului

```
copyinstr(SCARG(uap, msg), kmsg, 100, NULL);
```

partea de verificare a apelului este intenționat omisă pentru simplitate. În mod normal ea trebuie să existe, dar nu face obiectul laboratorului.

3 Funcții utilitare în kernel

Atenție, în kernel nu există biblioteca C standard sau alte funcții utilitare cu care suntem obișnuiți când scriem programe în userland. Cu toate astea, în kernelul de OpenBSD, există funcții similare cu cele din standardul de C.

`printf(9)` care se comportă similar cu funcția standard `printf(3)`. Diferența este că mesajul `printf(9)` apare în consola principală (`ttyC0`) și în logul `/var/log/messages`.

Pentru alocarea și eliberarea memoriei folosiți `malloc(9)` și `free(9)`. `malloc(9)` primește două argumente în plus: tipul memoriei alocat și cum să se efectueze alocarea. De exemplu pentru a aloca 10 bytes pentru un buffer temporar (folosit doar local în funcție) se folosește apelul

```
buf = malloc(10, M_TEMP, M_WAITOK);
```

ultimul argument anunțând că apelantul poate aștepta până se găsește memorie disponibilă. Când operațiile asupra lui `buf` s-au încheiat, acesta trebuie eliberat

```
free(buf, M_TEMP, 10);
```

4 Apelare din userland

Cel mai rapid mod de a apela o nouă funcție de sistem creată este cu ajutorul lui `syscall(2)`. Această funcție de sistem apelează o altă funcție de sistem cu ID-ul din primul argument. Restul argumentelor primite sunt pasate mai departe. De exemplu, pentru a apela noua funcție `khello` cu ID-ul 331 am folosi

```
syscall(331, "foo");
```

iar pentru a apela scrie `"Hello!"` pe ecran cu ajutorul funcției cunoscute `write(2)` am apela astfel

```
syscall(4, 1, "Hello!", 6);
```

unde 4 este ID-ul lui `write(2)` conform fișierului `/sys/kern/syscalls.master`.

Apelarea elegantă cu numele funcției de sistem se face prin declararea funcției în mai multe fișiere de tip `include` din sistem. Acest pas este lăsat drept exercițiu pentru acasă.

5 Sarcini de laborator

1. Compilați un kernel nou.
2. Adăugați o funcție de sistem nouă simplă care să afișeze ceva pe ecran și demonstrați că merge apelând-o dintr-un program. Exemplu apel `syscall(id-functie, "world")`. **Atenție**, trebuie să recompilați kernelul și să reporniți sistemul de operare cu kernelul nou!
3. Modificați funcția de sistem de mai devreme să copieze un număr dat de bytes dintr-un buffer sursă într-altul destinație. La ieșire funcția va scrie numărul de bytes copiat efectiv. Verificați intrările primite și semnalați eventualele erori. Exemplu apel `sz = kcp(src,dst,10)`.

Laboratorul 5

Comunicare Inter-Proces

1 Memoria Partajată

În mediile de dezvoltare care respectă standardul POSIX (toate variantele UNIX și în mare parte Windows), obiectele de memorie partajată se creează cu ajutorul funcției `shm_open(3)`

```
int shm_open(const char *path, int flags, mode_t mode);
```

având o semantică aproape identică cu funcția de sistem `open(2)` (vezi Laboratorul 2) motiv pentru care `shm_open(3)` este deobicei doar un wraper peste aceasta. Argumentul `path` este de fapt numele obiectului și nu o cale în sistemul de fișiere. Un apel tipic arată astfel

```
char shm_name[] = "myshm";
int shm_fd;

shm_fd = shm_open(shm_name, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR);
if (shm_fd < 0) {
        perror(NULL);
        return errno;
}
```

unde obiectul `"myshm"`, care dacă nu există este creat (`O_CREAT`), este deschis pentru scriere și citire (`O_RDWR`) oferind drepturi asupra lui doar utilizatorului care l-a creat (`S_IRUSR | S_IWUSR`, vezi Laboratorul 2 și `chmod(2)`). Rezultatul este un descriptor `shm_fd` pe care îl putem folosi mai departe în orice funcție ce manipulează obiecte cu ajutorul descriptorilor precum toate funcțiile de sistem sau din biblioteca standard de C pentru fișiere.

O dată creat primul pas este să îi definim dimensiunea cu ajutorul funcției de sistem `ftruncate(2)`

```
size_t shm_size = 1000;

if (ftruncate(shm_fd, shm_size) == -1) {
        perror(NULL);
        shm_unlink(shm_name);
        return errno;
}
```

Aceasta scurtează sau mărește obiectul asociat descriptorului dat conform noii dimensiuni primite în al doilea argument. În exemplul nostru mărește obiectul `shm_fd` de la 0 bytes la 1000.

Funcția `shm_unlink(3)` șterge obiectele create cu funcția `shm_open(3)` primind numele obiectului ca parametru. Aceasta este din nou o extindere firească de la funcția de sistem `unlink(2)` folosită în mod normal pentru a șterege fișiere de pe disk. Un apel tipic poate fii văzut în exemplul `ftruncate(2)` de mai devreme.

Memoria partajată se încarcă în spațiul procesului cu ajutorul funcției de sistem `mmap(2)`.

```
void * mmap(void *addr, size_t len, int prot, int flags,
            int fd, off_t offset);
```

Parametrii sunt

- `addr` – adresa la care să fie încărcată în proces (de obicei aici folosim 0 pentru a lăsa kernelul să decidă unde încarcă)
- `len` – dimensiunea memoriei încărcate
- `prot` – drepturile de acces (`PROT_READ` sau `PROT_WRITE` de obicei)
- `flags` – tipul de memorie (de obicei `MAP_SHARED` astfel încât modificările făcute de către proces să fie vizibile și în celelalte)
- `fd` – descriptorul obiectului de memorie
- `offset` – locul în obiectul de memorie partajată de la care să fie încărcat în spațiul procesului

iar, când se execută cu succes, rezultatul este un pointer către adresa din spațiul procesului la care a fost încărcat obiectul. Altfel, valoarea `MAP_FAILED` este întoarsă și `errno` este setat corespunzător.

Apelurile cele mai des întâlnite sunt de tipul

```
shm_ptr = mmap(0, shm_size, PROT_READ, MAP_SHARED, shm_fd, 0);
if (shm_ptr == MAP_FAILED) {
        perror(NULL);
        shm_unlink(shm_name);
        return errno;
}
```

unde `shm_ptr` va indica către **toată** zona de memorie (`shm_size`) aferentă descriptorului (`shm_fd`) care va fii doar citită (`PROT_READ`) și împărțită cu restul proceselor (`MAP_SHARED`), sau de tipul

```
shm_ptr = mmap(0, 100, PROT_WRITE, MAP_SHARED, shm_fd, 500);
if (shm_ptr == MAP_FAILED) {
        perror(NULL);
        shm_unlink(shm_name);
        return errno;
}
```

unde `shm_ptr` va indica către o **parte** de **100** bytes care începe de la byte-ul **500** din zona de memorie aferentă descriptorului (`shm_fd`) ce va fii doar scrisă (`PROT_WRITE`) și împărțită cu restul proceselor (`MAP_SHARED`).

ATENȚIE: în Linux dimensiunea trebuie să fie multiplu de pagini: `PAGE_SIZE`. Se poate obține și cu `getpagesize(2)`.

Când nu mai este nevoie de zona de memorie încărcată, se folosește funcția `munmap(2)`

```
int munmap(void *addr, size_t len);
```

care primește pointer-ul către zona încărcată în spațiul procesului și dimensiunea ca argumente. Pentru exemplele anterioare am folosi

```
munmap(shm_ptr, shm_size);
```

pentru când a fost încărcată **toată** zona, și

```
munmap(shm_ptr, 100)
```

pentru când a fost încărcată o **parte**.

2 Sarcini de laborator

1. Ipoteza Collatz spune că plecând de la orice număr $n \in \mathbb{N}$ dacă aplicăm următorul algoritm

$$n = \begin{cases} n/2 & \mod(n,2) = 0 \\ 3n+1 & \mod(n,2) \neq 0 \end{cases}$$

   Implementați un program care să testeze ipoteza Collatz pentru mai multe numere date folosind memorie partajată.

   Pornind de la un singur proces părinte, este creat câte un copil care se ocupă de un singur număr și scrie seria undeva în memoria partajată. Părintele va crea obiectul de memorie partajată folosind `shm_open(3)` și `ftruncate(2)` și pe urmă va încărca în memorie întreg spațiul pentru citirea rezultatelor cu `mmap(2)`.

O convenție trebuie stabilită între părinte și fii astfel încât fiecare copil să aibă acces exclusiv la o parte din memoria partajată unde își va scrie datele (ex. împărțim memoria în mod egal pentru fiecare copil). Astfel, fiecare copil va încărca doar zona dedicată lui pentru scriere folosind dimensiunea cuvenită și un deplasament nenul în `mmap(2)`. Părintele va aștepta să termine execuția fiecare copil după care va scrie pe ecran rezultatele obținute de fii săi.

Arătați că cerințele de sus sunt îndeplinite folosindu-vă de `getpid(2)` și `getppid(2)`. Exemplu:

```
$ ./shmcollatz 9 16 25 36
Starting parent 75383
Done Parent 75383 Me 59702
Done Parent 75383 Me 3281
Done Parent 75383 Me 33946
Done Parent 75383 Me 85263
9: 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4
2 1.
16: 16 8 4 2 1.
25: 25 76 38 19 58 29 88 44 22 11 34 17 52 26 13 40
20 10 5 16 8 4 2 1.
36: 36 18 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5
16 8 4 2 1.
Done Parent 96028 Me 75383
```

2. În programul anterior folosiți `shm_unlink(3)` și `munmap(2)` pentru a elibera resursele folosite.

Laboratorul 6

Fire de execuție

1 Crearea firelor de execuție

Laboratoarele anterioare au discutat despre crearea unor procese noi cu ajutorul funcțiilor de tip `fork(2)` sau `execve(2)`. Procesele noi create erau o copie fidelă a celui inițial, dar resuresele erau complet separate fiind necesare diferite mecanisme de comunicație inter-proces pentru a colabora.

Pentru ușurarea comunicării și sporirea performanței se pot folosi fire de execuție separate ale aceluiași proces. Acestea au avantajul că împart toate resursele și orice modificare făcută în spațiul procesului de un fir este instantaneu vizibilă tuturor celorlalte fără a apela la un mecanism exterior.

Dezavantajele apar o dată cu nevoia de a scrie și/sau citi concomitent aceiași zonă de memorie. În acest caz, concurența și drepturile de acces asupra resursei trebuie dictate de terți îngreunând astfel procesul de execuție și de multe ori introducând defecte subtile în program.

Firele de execuție, denumite thread în literatura de specialitate, sunt implementate în mediile POSIX prin variabile de tip `pthread_t`. Pentru a creea un thread nou se folosește funcția `pthread_create(3)`

```
int
pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine)(void *), void *arg);
```

care inițializează `thread` cu noul fir de execuție lansat prin apelarea funcției `start_routine` cu argumentele oferite de `arg`. **Atenție**, diferit de procesele create cu `fork(2)`, un thread nou pornește execuția de la o funcție dată.

La inițializare, se pot particulariza anumite detalii privind noul thread (ex. dimensiunea stivei, chestiuni de securitate etc.). Pe parcursul laboratorului vom folosi atributele implicit setate de sistemul de operare, semnalând aceasta prin folosirea valorii `NULL` pentru `attr`.

Un apel tipic pentru lansarea unui nou fir de execuție este

```
pthread_t thr;
if (pthread_create(&thr, NULL, hello, "world!")) {
        perror(NULL);
        return errno;
}
```

unde funcția `hello` trebuie să respecte prototipul `start_routine` de mai devreme

```
void *
hello(void *v)
{
        char *who = (char *)v;
        printf("Hello, %s!", who);
        return NULL;
}
```

Pentru a aștepta finalizarea execuției unui thread se folosește `pthread_join(3)`

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

care, diferit de `wait(2)`, așteaptă explicit firul de execuție din variabila `thread`. Dacă `value_ptr` nu este `NULL`, atunci `pthread_join` va pune la adresa indicată rezultatul funcției `start_routine`. Un apel tipic este

```
if (pthread_join(thr, &result)) {
        perror(NULL);
        return errno;
}
```

În mediile POSIX funcționalitatea thread se găsește într-o bibliotecă separată numită `libpthread`. Astfel la compilare este nevoie să specificăm explicit această legătură

```
$ cc hello.c -o hello -pthread
```

2 Sarcini de laborator

1. Scrieți un program care primește un șir de caractere la intrare ale cărui caractere le copiază în în ordine inversă și le salvează într-un șir separat. Operație de inversare va avea loc într-un thread separat. Rezultatul va fi obținut cu ajutorul funcției `pthread_join`. Exemplu

   ```
   $ ./strrev hello
   olleh
   ```

2. Scrieți un program care să calculeze produsul a două matrice date (de dimensiuni compatibile) unde fiecare element al matricei rezultate este calculat de către un thread distinct.

   Reamintim că date $\boldsymbol{A} \in \mathbb{R}^{m \times p}$ și $\boldsymbol{B} \in \mathbb{R}^{p \times n}$, elementul c_{ij} al matricei $\boldsymbol{C} \in \mathbb{R}^{m \times n}$ este calculat cu ajutorul formulei

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}.$$

Laboratorul 7

Sincronizare

1 Excluziune mutuală

Adesea în programele cu mai multe fire de execuție și procese avem nevoie ca o singură entitate să execute un număr de instrucțiuni la un moment de timp dat. Această zonă în care are voie un singur proces sau thread se numește zonă critică (critical section).

La laborator vom folosi exclusiv pentru paralelism fire de execuție, dar conceptele folosite se pot aplica la fel de bine pentru situația în care avem de-a face cu mai mult procese.

Obiectul cel mai des folosit pentru asigurarea accesului exclusiv într-o zonă critică este mutex-ul (prescurtat de la **mut**ual **ex**clusive). În bibliotecile de sistem ce implementează standardul POSIX, un obiect de tip mutex este creat cu ajutorul funcției `pthread_mutex_init(3)`

```
int pthread_mutex_init(pthread_mutex_t *mutex,
 const pthread_mutexattr_t *attr);
```

care inițializează un `mutex` cu atributele `attr` (la laborator se va folosi `NULL` pentru a obține atributele implicite similar cum am procedat pentru `pthread`). De obicei variabila mutex se pune fie în memoria globală, fie într-o structură accesibilă tuturor firelor de execuție după care se inițializează în `main()` sau undeva înainte de a fii folosit.

```
pthread_mutex_t mtx;
int main()
{
        /* ... */
        if (pthread_mutex_init(&mtx, NULL)) {
                perror(NULL);
                return errno;
        }
        /* ... */
}
```

Un mutex se poate afla în două stări: fie închis (locked), fie deschis (unlocked). Când mutexul este închis înseamnă că un thread deține dreptul exclusiv de execuție asupra zonei critice până ce decide să renunțe la acest drept.

Pentru a obține un mutex (i.e. pentru a-l închide) se folosește funcția `pthread_mutex_lock(3)` iar pentru a-l elibera se folosește `pthread_mutex_unlock(3)`. Ele sunt folosite tipic în tandem, demarcând zona critică

```
pthread_mutex_lock(&mtx);
count++;        /* critical section */
pthread_mutex_unlock(&mtx);
```

Atenție, funcția `pthread_mutex_lock(3)` nu se termină de executat până nu obține mutexul, blocând astfel firul de execuție ce a apelat-o!

La sfârșit, când nu mai avem nevoie de obiectul de tip mutex, eliberăm resursele ocupate cu `pthread_mutex_destroy(3)`

```
pthread_mutex_destroy(&mtx);
```

2 Semafoare

Semafoarele sunt similare obiectelor de tip mutex, dar pot face față unor scenarii mai sofisticate de sincronizare.

În esență, un semafor este o variabilă S inițializată cu o valoare întreagă care este manipulată exclusiv cu ajutorul a două funcții: `wait` și `post` (sau `signal`). În principiu, ele scad, respectiv cresc, cu o unitate valoarea lui S.

Condiția principală ca obiectele de tip semafor să funcționeze corect este ca funcțiile de mai sus să se execute fără a fii întrerupte (să se execute atomic). Astfel putem observa că un obiect de tip mutex este defapt un caz particular de semafor în care $S = 1$.

În mediile POSIX semafoarele sunt inițializate cu `sem_init(3)`

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

care setează valoarea inițială a semaforului `sem` cu $S =$ `value`. Variabila `pshared` este folosită pentru a semnala dacă vrem să folosim semaforul în cadrul mai multor procese. Nefiind cazul în cadrul laboratorului, vom folosi tot timpul `pshared` $= 0$. Apelul tipic este similar cu cel în cazul obiectelor mutex

```
sem_t sem;
int main()
{
        /* ... */
        if (sem_init(&sem, 0, S)) {
                perror(NULL);
                return errno;
        }
        /* ... */
}
```

unde S este valoarea inițială aleasă.

Funcția `sem_wait(3)` scade valoarea lui S cu o unitate și poate fii executată cu siguranță în medii de lucru paralele.

```
if (sem_wait(&sem)) {
        perror(NULL);
        return errno;
}
```

Atenție, dacă $S = 0$ atunci funcția așteaptă ca valoarea să crească înainte de a o scădea blocând astfel firul apelant!

Similar, pentru a crește valoarea lui S este folosită funcția `sem_post(3)`

```
if (sem_post(&sem)) {
        perror(NULL);
        return errno;
}
```

care, după incrementare $S = S + 1$, verifică dacă sunt thread-uri blocate de semafor și eliberează thread-ul care așteaptă de cel mai mult timp în coadă. Acesta va relua execuția din punctul în care a apelat `sem_wait(3)`.

Când nu mai este folosit, semaforul este eliberat cu ajutorul funcției `sem_destroy(3)`

```
sem_destroy(&sem);
```

3 Sarcini de laborator

1. Scrieți un program care gestionează accesul la un număr finit de resurse. Mai multe fire de execuție pot cere concomitent o parte din resurse pe care le vor da înapoi o dată ce nu le mai sunt utile. Fie numărul maxim de resurse dat

   ```
   #define MAX_RESOURCES 5
   int available_resources = MAX_RESOURCES;
   ```

   Când un thread dorește să oțină un număr de resurse acesta apelează `decrease_count`.

   ```
   int decrease_count(int count)
   {
           if (available_resources < count)
                   return -1;
           else
                   available_resources -= count;
           return 0;
   }
   ```

   iar când resursele nu-i mai sunt necesare apelează `increase_count`

```
int increase_count(int count)
{
        available_resources += count;
        return 0;
}
```

Funcțiile de mai sus prezintă mai multe defecte într-un mediu de execuție paralel printre care și un **race condition**. Modificați funcțiile și rezolvați race condition-ul folosind obiecte de tip mutex. Arătați că modificările dumneavoastră sunt corecte cu ajutorul unui program care pornește mai multe thread-uri ce consumă un număr diferit de resurse fiecare. De exemplu:

```
$ ./count
MAX_RESOURCES=5
Got 2 resources 3 remaining
Released 2 resources 5 remaining
Got 2 resources 3 remaining
Released 2 resources 5 remaining
Got 1 resources 4 remaining
Released 1 resources 5 remaining
Got 3 resources 2 remaining
Got 2 resources 0 remaining
Released 3 resources 3 remaining
Released 2 resources 5 remaining
```

2. Scrieți un program care să sincronizeze execuția a N fire de execuție construind un obiect de tip barieră. Bariera va fii inițializată folosind `init(N)` și fiecare thread va apela `barrier_point()` când va ajunge în dreptul barierei. Când funcția este apelată a N-a oară, aceasta pornește execuția tuturor firelor în așteptare.
   Verificați rezultatele dumneavoastră cu ajutorul unui program care pornește mai multe thread-uri ce se folosesc de barieră pentru a-și sincroniza execuția.

   Funcția executată de fiecare fir poate avea următoarea formă

```
void * tfun(void *v)
{
        int *tid = (int *)v;

        printf("%d reached the barrier\n", *tid);
        barrier_point();
        printf("%d passed the barrier\n", *tid);

        free(tid);
        return NULL;
}
```

unde `tid` este numărul threadului pornit. Astfel, o instanță cu 5 fire de execuție ar afișa

```
$ ./barrier
NTHRS=5
0 reached the barrier
1 reached the barrier
2 reached the barrier
3 reached the barrier
4 reached the barrier
4 passed the barrier
0 passed the barrier
1 passed the barrier
3 passed the barrier
2 passed the barrier
```

Indiciu, pentru a implementa obiectul de tip barieră folosiți un mutex pentru contorizarea firelor ajunse la barieră și un semafor pentru a aștepta la barieră.