

Procese Unix descrise in C: fork, exec, exit, wait, system

Contents

1. Standardul POSIX de gestiune a erorilor în apeluri sistem: errno	1
2. Principalele apeluri sistem Unix care operează cu procese	1
3. Utilizări simple fork exit, wait	2
4. Utilizări simple execl, execlp, execv, system	5
5. Un program care compileaza și rulează alt program	6
6. Capitalizarea cuvintelor dintr-o listă de fișiere text	7
7. Câte perechi de argumente au suma un număr par?	8
8. Probleme propuse	10

1. Standardul POSIX de gestiune a erorilor în apeluri sistem: errno

Marea majoritate a funcțiilor C și practic toate apelurile sistem Unix întorc un rezultat care "spune" dacă funcția/apelul s-a derulat normal sau dacă a apărut o situație deosebită. În caz de eșec funcția / apelul sistem întoarce fie un întreg nenul (valoarea 0 este rezervată pentru succes), fie un pointer NULL etc. Metodologic, se recomandă **SA SE APELEZE**:

```
if (functie( - - - ) == SUCCES) { tratare normala }
else { tratare situatie de derulare anormala }
```

NU (așa cum din comoditate apelează mulți programatori) :

```
functie( - - - ); tratare normala
```

Evident forma a doua este mai scurtă, dar nu sunt tratate situațiile de excepție.

Pentru o abordare unitară a acestor tratamente, standardul POSIX oferă prin `#include <errno.h>` o variabilă întreagă **errno**, (care nu trebuie declarată) a carei valoare este setată de sistem nunai în caz de derulare anormală a apelului! **La o situație de derulare anormală sistemul fixează o valoare nenulă ce indică cauza erorii.** Pentru detalii vezi man `errno`, precum și lista completă a cazurilor de erori, aflată de exemplu la <http://www.virtsync.com/c-error-codes-include-errno>

La apelul cu succes al unei funcții sistem errno nu se setează la 0! Pentru a se vedea și în clar eroarea depistată se pot folosi funcțiile `strerror` și `perror` dau detalii pentru fiecare valoare a lui `errno`:

```
#include <errno.h>
- - -
if (functie( - - - ) == SUCCES) { tratare normala }
else { perror("Eroarea depistata este:");
      // sau printf("Eroarea depistata este:%s", strerror(errno); }
```

2. Principalele apeluri sistem Unix care operează cu procese

Tabelul următor prezintă sintaxele principalelor apeluri sistem Unix care operează cu procese:

Funcții specifice proceselor	Funcții specifice comunicării între procese
<code>fork()</code>	<code>pipe(f)</code>
<code>exit(n)</code>	<code>mkfifo(nume, drepturi)</code>
<code>wait(p)</code>	<code>FILE *popen(c, "r w")</code>
<code>exec*(c, lc)</code>	<code>pclose(FILE *)</code>
<code>system(c)</code>	<code>dup2(fo, fn)</code>

Prototipurile lor sunt descrise, de regula, în `<unistd.h>` Parametrii sunt:

- `n` este întreg – codul de retur cu care se termină procesul;
- `p` este un pointer la un întreg unde fiul întoarce codul de retur (extras cu funcția `WEXITSTATUS`);
- `c` este o comandă Unix;
- `lc` este linia de comandă (comanda `c` urmată de argumentele liniei de comandă);
- `f` este un tablou de doi întregi – descriptori de citire / scriere din / în pipe;
- `nume` este numele (de pe disc) al fișierului FIFO, iar `drepturi` sunt drepturile de acces la acesta;
- `fo` și `fn` descriptori de fișiere: `fo` deschis în program cu `open`, `fn` poziția în care e duplicat `fo`.

În caz de eșec, funcțiile întorc -1 (NULL la `popen`) și poziționează `errno` se depistează ce eroare a apărut.

Funcțiile `system` și `popen` au comanda completă (un string), interpretabilă de shell: aceste funcții lansează mai întâi un shell, apoi în acesta lansează comanda `c`. Această lansare se face simplu folosind un apel sistem `execl("/bin/sh", "sh", "-c", c, NULL);`

Comanda `c` din apelurile `exec*` **NU permite specificări folosite uzual în liniile shell**. Astfel, în `c` NU trebuie să apară specificări generice de fișiere, redirectări de intrări / ieșiri standard, variabile shell, captări de ieșiri prin construcții `` - - comanda - - `` etc. Dacă totuși se dorește acest lucru, trebuie să se lanseze, ca mai sus, interpretorul `sh` cu opțiunea `-c` și apoi să se specifice comanda.

3. Utilizări simple `fork`, `exit`, `wait`

Vom prezenta și discuta două exemple de programe care utilizează apelurile sistem `fork`, `exit`, `wait`.

Să considerăm **programul f1.c** căruia i-am numerotat liniile sursă:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5  int main() {
6      int p, i;
7      p=fork();
8      if (p == -1) {perror("fork imposibil!"); exit(1);}
9      if (p == 0) {
10         for (i = 0; i < 10; i++)
11             printf("Fiu: i=%d pid=%d, ppid=%d\n", i, getpid(), getppid());
12         exit(0);
13     } else {
14         for (i = 0; i < 10; i++)
15             printf("Parinte: i=%d pid=%d ppid=%d\n", i, getpid(), getppid());
16         wait(0);
17     }
18     printf("Terminat; pid=%d ppid=%d\n", getpid(), getppid());
19 }
```

Vom analiza comportamentul acestui program în diverse situații, făcând o serie de modificări în această sursă.

Rularea în forma inițială: Sunt afișate 21 linii: 10 ale fiului de la linia 11 cu pidul lui și al părintelui. 11 ale fiului, 10 de la linia 15 și ultima de la linia 18. Părintele părintelui este pidul shell. Este posibil ca ordinea primelor 20 de linii să apară amestecate, linii ale fiului și liniile ale părintelui. Dacă la linia 10 și la linia 14 se înlocuiește 10 cu 1000, se vor afișa 2001 linii iar amestecarea între liniile fiului și ale părintelui va fi mai evidentă.

Comentarea liniei 12: Procesul fiu se termină la linia 18, ca și părintele. Se vor tipări 22 linii, linia 18 se va tipări de două ori: odată de părinte și odată de fiu.

Comentarea liniei 16: Părintele nu mai așteaptă terminarea fiului și acesta din urmă rămâne în starea zombie. Se tipăresc cele 21 de linii ca în primul caz. O observație interesantă: dacă ieșirea programului se redirectează într-un fișier pe disc, apar cele 21 linii. În schimb, dacă ieșirea se face direct pe terminal, apar doar liniile fiului. De ce oare? Rămâne un TO DO pentru studenți.

Comentarea liniilor 12 și 16: Se tipăresc 22 linii, cu aceeași observație de mai sus, de la comentarea liniei 16. Aici recomandăm modificări ale numărului liniilor tipărite de fiu (linia 10) și a celor tipărite de părinte (linia 14). Se vor vedea efecte interesante.

Să considerăm **programul f2.c**:

```
main() { fork(); if (fork()) {fork();} printf("Salut\n"); }
```

Care este efectul execuției acestui program? (Acoladele nu sunt necesare, dar le-am pus pentru a evidenția mai bine corpul lui `if`). Să facem o primă analiză:

- Primul `fork` naște un proces fiu. Ambele procese au de executat secvența: `if (fork()) fork(); printf("Salut\n");`
- Condiția `fork` din `if` mai naște câte un proces fiu cărora le rămâne de făcut doar `printf("Salut\n");` În același timp, cele două procese care evaluează `if` mai au de făcut `{fork();} printf("Salut\n");` Până aici avem patru procese.
- Fiecare `fork` dintre acolade mai naște câte un proces fiu căruia îi mai rămâne de făcut `printf("Salut\n");` Avem încă două procese în plus.
- În concluzie, avem șase (6) procese care au de executat `printf("Salut\n");` În consecință, se va tipări de 6 ori Salut.

Merită să studiem mai atent acest exemplu. Principala carență a lui este aceea că nici un părinte care naște un fiu nu așteaptă terminarea lui prin `wait`. Consecința, vor rămâne câteva procese în starea zombie.

Pentru a aprofunda analiza, să rescriem puțin programul `f2.c`:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
    printf("START: pid=%d ppid=%d\n", getpid(), getppid());
    int i=-2, j=-2, k=-2;
    i=fork();
    if (j=fork())
        {k=fork();}
    printf("Salut pid=%d ppid=%d i=%d j=%d k=%d\n",getpid(),getppid(),i,j,k);
}
```

În fapt, am reținut în variabilele *i*, *j*, *k* valorile PID-urilor create pe parcursul execuției. Rezultatul execuției este:

```
START: pid=3998 ppid=3810
Salut pid=3998 ppid=3810 i=3999 j=4000 k=4001
florin@ubuntu:~/c$ Salut pid=4001 ppid=1700 i=3999 j=4000 k=0
Salut pid=4000 ppid=1700 i=3999 j=0 k=-2
Salut pid=3999 ppid=1700 i=0 j=4002 k=4003
Salut pid=4003 ppid=1700 i=0 j=4002 k=0
Salut pid=4002 ppid=1700 i=0 j=0 k=-2
```

Să analizăm ordinea în care se execută aceste instrucțiuni:

- 3810 este PID-ul shell care afișează promptul, iar 3998 este PID-ul programului inițial.
- Procesul 3998 crează fiul *i* cu PID-ul 3999, fiul *j* cu PID-ul 4000 și fiul *k* cu PID-ul 4001. Apoi își face tipărirea și se termină - se vede tipărirea promptului.
- Cele trei procese 3999, 4000 și 4001 rămân active dar sunt în starea zombie (PPID-ul lor este 1700).
- Procesul 4001 preia controlul procesorului, valorile *i* și *j* sunt moștenite de la 3998, iar *k* = 0 fiind vorba de fork în fiu, face tipărirea și se termină.
- Procesul 4000 preia controlul procesorului, valorile *i* și *k* sunt moștenite de la 3998 - *i* creat, *k* încă necreat, iar *j* = 0 fiind vorba de fork în fiu, face tipărirea și se termină.
- Procesul 3999 preia controlul procesorului, *i* = 0 fiind vorba de fork în fiu, crează fiul *j* cu PID-ul 4002 și fiul *k* cu PID-ul 4003. Apoi își face tipărirea și se termină.
- Procesul 4003 preia controlul procesorului, valorile *i* și *j* sunt moștenite de la 3999, iar *k* = 0 fiind vorba de fork în fiu. Apoi își face tipărirea și se termină.
- Procesul 4002 preia controlul procesorului, valorile *i* și *k* sunt moștenite de la 3999, iar *j* = 0 fiind vorba de fork în fiu. Apoi își face tipărirea și se termină.

Tabelul următor prezintă cele 6 procese: ce valori moștenesc de la părinte, ce cod mai au de executat și ce valori finale au (ce tipăresc).

PID 3998 PPID 3810 <i>i</i> =-2 <i>j</i> =-2 <i>k</i> =-2 <i>i</i> =fork(); if (<i>j</i> =fork()) { <i>k</i> =fork();} printf - - - <i>i</i> =3999 <i>j</i> =4000 <i>k</i> =4001					
	PID 4001 PPID 3998 <i>i</i> =3999 <i>j</i> =4000 <i>k</i> =0 printf - - - <i>i</i> =3999 <i>j</i> =4000 <i>k</i> =0	PID 4000 PPID 3998 <i>i</i> =3999 <i>j</i> =0 <i>k</i> =-2 printf - - - <i>i</i> =3999 <i>j</i> =0 <i>k</i> =-2	PID 3999 PPID 3998 <i>i</i> =0 <i>j</i> =-2 <i>k</i> =-2 if (<i>j</i> =fork()) { <i>k</i> =fork();} printf - - - <i>i</i> =0 <i>j</i> =4002 <i>k</i> =4003		
				PID 4003 PPID 3999 <i>i</i> =0 <i>j</i> =4002 <i>k</i> =0 printf - - - <i>i</i> =0 <i>j</i> =4002 <i>k</i> =0	PID 4002 PPID 3999 <i>i</i> =0 <i>j</i> =0 <i>k</i> =-2 printf - - - <i>i</i> =0 <i>j</i> =0 <i>k</i> =-2

În acest tabel, valoarea PPID este cea reală a părintelui creator, deși la momentul terminării fiului părintele nu mai există, așa că procesul intră în starea zombie.

Pentru a evita starea acumularea de procese în starea zombie, se poate folosi, spre exemplu, secvența:

```
- - -
# include <signal.h>
int main() {
    signal(SIGCHLD, SIG_IGN);
- - -
```

În acest fel se cere ignorarea trimiterii de către fiu a semnalului SIGCHLD, pe care părintele ar trebui să îl primească (să fie în viață), să îl trateze cu un wait. Prin această ignorare, procesul fiu este sters din sistem imediat după terminarea lui.

4. Utilizări simple execl, execlp, execv, system

Urmatoarele două programe, deși diferite, au același efect. Toate trei folosesc o comandă de tip `exec`, spre a lansa din ea comandă shell:

```
ls -l
```

Programul 1:

```
#include <stdio.h>
#include <unistd.h>
int main() {
    char* argv[3];
    argv[0] = "/bin/ls";
    argv[1] = "-l";
    argv[2] = NULL;
    execv("/bin/ls", argv);
}
```

Aici se pregătește linia de comandă în vectorul `argv` spre a o lansa cu `execv`.

Programul 2:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h> // trebuie pentru system
int main() {
    //execl("/bin/ls", "/bin/ls", "-l", NULL);
    // execlp("ls", "ls", "-l", NULL);
    // execl("/bin/ls", "/bin/ls", "-l", "p1.c", "execl.c", "fork1.c", "xx", NULL);
    // execl("/bin/ls", "/bin/ls", "-l", "*.c", NULL);
    system("ls -l *.c");
}
```

Aici se execută, pe rând, numai una dintre cele 5 linii, comentând pe celelalte 4. Ce se va întâmpla?

- Primul `execl` lansează `ls` prin cale absolută și are același efect ca și programul 1.
- Al doilea lansează `ls` prin directoarele din `PATH`, efectul este același.
- Al treilea cere `ls` pentru o listă de fișiere. Pentru cele care nu există, se dă mesajul: `/bin/ls: cannot access 'xx': No such file or directory` (în loc de `xx` apar numele fișierelor inexistente);
- Al patrulea `exec` va da mesajul: `/bin/ls: cannot access *.c: No such file or directory`
Nu este interpretat așa cum ne-am aștepta! De ce? Din cauza faptului că specificarea `*.c` reprezintă o specificare generică de fișier, dar numai shell "știe" acest lucru și el (shell) înlocuiește această specificare, în cadrul uneia dintre etapele de tratare a liniei de comandă. La fel stau lucrurile

cu evaluarea variabilelor de mediu, \${---}, inlocuirea dintre apostroafele inverse ` --- `, redirectarea I/O standard etc.

- Apelul `system` are efectul așteptat, făcând rezumatul tuturor fișierelor de tip `c` din directorul curent.

Să ne oprim puțin asupra funcției `system`. Ea crează prin `fork` un proces fiu, în care lansează comanda `execl("/bin/sh", "sh", "-c", c, NULL)` așa cum am arătat mai sus și întoarce codul de retur cu care s-a terminat execuția comenzii. Doritorii pot să vadă sursa `system.c`, care este o funcție simplă, de maximum 100 linii în care se includ comentariile, tratările cu `errno` ale posibilelor erori și manevrarea unor semnale specifice. Sursa poate fi găsită la: <http://man7.org/tlpi/code/online/dist/procexec/system.c>

5. Un program care compileaza și rulează alt program

Exemplul care urmeaza are acelasi efect ca si scriptul `sh`:

```
#!/bin/sh
if gcc -o ceva $1
then ./ceva $*
else echo "Erori de compilare"
fi
```

Noi nu il vom implementa în `sh`, ci vom folosi programul `compilerun.c`.

```
// Similar cu scriptul shell:
// #!/bin/sh
// if gcc -o ceva $1; then ./ceva $*
//     else echo "Erori de compilare"
// fi
#include<stdio.h>
#include<unistd.h>
#include <stdlib.h>
#include<string.h>
#include <sys/wait.h>
int main(int argc, char* argv[]) {
    char comp[200];
    char* run[100];
    int i;
    strcpy(comp, "gcc -o ceva ");
    strcat(comp, argv[1]); // Fabricat comanda de compilare
    if (WEXITSTATUS(system(comp)) == 0) {
        run[0] = "./ceva";
        for (i = 1; argv[i]; i++) run[i] = argv[i];
        run[i] = NULL; // Fabricat comanda pentru execv
        execv("./ceva", run);
    }
    printf("Erori de compilare\n");
}
```

Compilarea lui se face

```
gcc -o compilerun compilerun.c
```

Executia se face, de exemplu, prin

```
./compilerun argvenvp.c a b c
```

Ceafect, daca compilarea sursei argument (argvenvp.c) este corecta, atunci compilatorul gcc creeaza fisierul ceva si intoarce cod de retur 0, dupa ceva este lansat prin execv. Daca esueaza compilarea, se va tipari doar mesajul "Erori de compilare".

Am ales ca și exemplu de program argvenvp.c:

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[], char *envp[]) {
    int i;
    printf("Argumentele:\n");
    for (i = 1; argv[i]; i++) printf("%s\n", argv[i]);
    printf("Cateva variabile de mediu:\n");
    for (i = 0; envp[i]; i++)
        if (strcmp("HOME", envp[i], 4)==0 || strcmp("LOGNAME", envp[i], 7)==0)
            printf("%s\n", envp[i]);
}
```

Secvența de execuție este:

```
florin@ubuntu:~/c$ gcc -o compilerun compilerun.c
florin@ubuntu:~/c$ ./compilerun argvenvp.c a b c
Argumentele:
argvenvp.c
a
b
c
Cateva variabile de mediu:
HOME=/home/florin
LOGNAME=florin
florin@ubuntu:~/c$
```

6. Capitalizarea cuvintelor dintr-o listă de fișiere text

Se cere un program care primește la linia de comandă o listă de fișiere text. Se cere ca toate aceste fișiere să fie transformate în altele, cu același conținut, dar în care fiecare cuvânt să înceapă cu literă mare. Se vor lansa procese paralele pentru prelucrarea simultană a tuturor fișierelor.

Pentru aceasta, vom crea mai întâi un program cu numele cap din sursa cap.c. Acesta primește la linia de comandă numele a două fișiere text, primul de intrare, al doilea de ieșire cu cuvintele capitalizate:

```
#include <stdio.h>
#include <string.h>
#define MAXLINIE 100
main(int argc, char* argv[]) {
    printf("Fiu: %d ...> %s %s\n", getpid(), argv[1], argv[2]);
    FILE *fi, *fo;
    char linie[MAXLINIE], *p;
    fi = fopen(argv[1], "r");
    fo = fopen(argv[2], "w");
    for ( ; ; ) {
        p = fgets(linie, MAXLINIE, fi);
        linie[MAXLINIE-1] = '\0';
        if (p == NULL) break;
        if (strlen(linie) == 0) continue;
        linie[0] = toupper(linie[0]); // Pentru cuvantul care incepe in coloana 0
        for (p = linie; ; ) {
            p = strstr(p, " ");
            if (!p) break;
            *p = toupper(*p);
            p++;
        }
        fprintf(fo, "%s", linie);
        if (p != linie) fprintf(fo, " ");
        if (p == linie) fprintf(fo, "\n");
    }
}
```

```

        if (p == NULL) break;
        p++;
        if (*p == '\n') break;
        *p = toupper(*p); // Caracterul de dupa spatiu este facut litera mare
    }
    fprintf(fo, "%s", linie);
}
fclose(fo);
fclose(fi);
}

```

Al doilea program, numit `master.c` va crea câte un proces pentru fiecare nume de fișier primit la linia de comandă și în acel proces va lansa `cap fi fi.CAPIT`

```

#include <stdio.h>
#include <string.h>
#include <unistd.h>
main(int argc, char* argv[]) {
    int i, pid;
    char argvFiu[200];
    for (i=1; argv[i]; i++) {
        pid = fork();
        if (pid == 0) {
            strcpy(argvFiu, argv[i]);
            strcat(argvFiu, ".CAPIT");
            execl("./cap", "./cap", argv[i], argvFiu, NULL);
        } else
            printf("Parinte, lansat fiul: %d ...> %s %s \n", pid, argv[i], argvFiu);
    }
    for (i=1; argv[i]; i++) wait(NULL);
    printf("Lansat simultan %d procese de capitalizare\n", argc - 1);
}

```

Compilari:

```

>gcc -o cap cap.c
>gcc -o master master.c

```

Lansare `master f1 f2 ... fi ... fn`

7. Câte perechi de argumente au suma un număr par?

La linia de comandă se dau n perechi de argumente despre care se presupune ca sunt numere întregi și positive. Se cere numărul de perechi care au suma un număr par, numărul de perechi ce au suma număr impar și numărul de perechi în care cel puțin unul dintre argumente nu este număr strict pozitiv.

Rezolvarea: În procesul părinte se va crea câte un process fiu pentru fiecare pereche. Oricare dintre fii întoarce codul de retur:

- 0 dacă perechea are suma pară,
- 1 dacă suma este impară,
- 2 dacă unul dintre argumente este nul sau nenumeric.

Parintele așteaptă terminarea fiilor și din codurile de retur întoarce de aceștia va afișa rezultatul cerut.

Vom da doua solutii:

1. Solutia 1 cu textul complet într-un singur fisier sursa
2. Solutia 2 cu doua texte sursa si unul să îl apeleze pe celalalt prin `exec`.

Solutia 1:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
main(int argc, char* argv[]) {
    int pare = 0, impare = 0, nenum = 0, i, n1, n2;
    for (i = 1; i < argc-1; i += 2) {
        if (fork() == 0) {
            n1 = atoi(argv[i]); // atoi intoarce 0
            n2 = atoi(argv[i+1]); // si la nenumeric
            if (n1 == 0 || n2 == 0) exit(2);
            exit ((n1 + n2) % 2);
        }
    }
    for (i = 1; i < argc-1; i += 2) {
        wait(&n1);
        switch (WEXITSTATUS(n1)) {
            case 0: pare++; break;
            case 1: impare++; break;
            default: nenum++;
        }
    }
    printf("Pare %d, impare %d, nenum %d\n",pare, impare, nenum);
}
```

Solutia 2:

Se creaza programul `par.c` care primește la linia de comandă o pereche de argumente. Din această sursă se va constii prin `gcc -o par par.c` executabilul `par`:

```
main(int argc, char *argv[]) {
    int n1, n2;
    n1 = atoi(argv[1]); // atoi intoarce 0
    n2 = atoi(argv[2]); // si la nenumeric
    if (n1 == 0 || n2 == 0) exit(2);
    exit ((n1 + n2) % 2);
}
```

Se creaza programul `master.c` care primește la linia de comandă `n` perechi de argumente. El va crea `n` procese fii și în fiecare va lansa prin `exec` programul `par`. Din aceasta sursa se va constii prin `gcc -o master master.c` executabilul `master`:

```
main(int argc, char* argv[]) {
    int pare = 0, impare = 0, nenum = 0, i, n1, n2;
    for (i = 1; i < argc-1; i += 2) {
        if (fork() == 0) {
            execl("./par", "./par", argv[i], argv[i+1], NULL);
        }
    }
    for (i = 1; i < argc-1; i += 2) {
        wait(&n1);
        switch (WEXITSTATUS(n1)) {
            case 0: pare++; break;
            case 1: impare++; break;
            default: nenum++;
        }
    }
    printf("Pare %d, impare %d, nenum %d\n",pare, impare, nenum);
}
```

Intrebare la ambele solutii: Ce se întâmplă dacă `wait` și `switch` nu sunt plasate în cicluri `for` succesive ci în același `for` care crează procesele fii?

8. Probleme propuse

1. Programul apelat compară două sau mai multe numere primite ca argumente și returnează cod 0 dacă toate sunt egale, 1 altfel. Programul apelant citește niste numere și spune dacă sunt egale.
2. Programul apelat primește ca argumente un nume de fișier și un șir de caractere și scrie în fișier șirul oglindit. Programul apelat citește niste șiruri de caractere și concatenează oglindirile lor.
3. Programul apelat primește ca argumente niste numere și returnează cod 0 dacă produsul lor este pozitiv, 1 dacă e negativ și 2 dacă e nul. Programul apelant citește un șir de numere și afișează dacă produsul lor este pozitiv, negativ sau zero.
4. Programul apelat primește ca argumente două numere naturale și un nume de fișier și scrie în fișier cel mai mic multiplu comun al numerelor. Programul apelant citește un șir de numere naturale și afișează cel mai mic multiplu comun al lor.
5. Programul apelat primește ca argumente două numere naturale și un nume de fișier și scrie în fișier cel mai mare divizor comun al numerelor. Programul apelant citește un șir de numere naturale și afișează cel mai mare divizor comun al lor.
6. Programul apelat primește ca argumente două numere și un nume de fișier și scrie în fișier produsul numerelor. Programul apelant citește un șir de numere și afișează produsul lor.
7. Programul apelat primește ca argumente trei nume de fișiere, primele două conținând câte un șir crescător de numere întregi, și scrie în al treilea fișier rezultatul interclasării șirurilor din primele două fișiere. Programul apelant citește un șir de numere întregi, le sortează și scrie rezultatul sortării.
8. Programul apelat primește ca argumente un nume de fișier și niste șiruri de caractere și le concatenează, rezultatul fiind scris în fișierul dat ca prim argument. Programul apelat citește niste șiruri de caractere și le concatenează.
9. Programul apelat primește ca argumente niste numere și returnează cod 0 dacă suma lor este pară și 1 altfel. Programul apelant citește un șir de numere și afișează dacă suma lor este pară sau nu.
10. Programul apelat primește ca argumente două numere și returnează cod 0 dacă sunt prime între ele și 1 altfel. Programul apelant citește un șir de numere și determină dacă sunt două câte două prime între ele.
11. Programul apelat primește ca argument un număr natural și returnează cod 0 dacă este prim și 1 altfel. Programul principal citește un număr n și afișează numerele prime mai mici sau egale cu n .
12. Programul apelat primește ca argumente două nume de fișier și adaugă conținutul primului fișier la al doilea fișier. Programul apelant primește un șir de nume de fișiere și concatenează primele fișiere punând rezultatul în ultimul fișier.
13. Programul apelat primește ca argumente două numere și un nume de fișier și adaugă în fișier toate numerele prime cuprinse între cele două numere date. Programul apelant citește un număr și scrie toate numerele prime mai mici decât numărul dat, apelând celălalt program pentru intervale de cel mult 10 numere.

14. Programul apelat primește ca argumente două numere și un nume de fișier și scrie în fișier suma numerelor. Programul apelant citește un sir de numere și afișează suma lor.
15. Programul apelat primește ca argumente două sau mai multe numere și returnează cod 0 dacă sunt două câte două prime între ele, 1 altfel. Programul apelant citește niste numere și spune dacă sunt două câte două prime între ele.