3. Procese Unix (în C): fork, exec, exit, wait system, signals

Contents

3.1. Gestiunea erorilor în apeluri de funcţii şi în apeluri sistem: errno

Metodologic, **se recomandă ca funcţiile C definite de utilizator. dacă este posibil, să întoarcă o valoare care să "informeze" modul de derulare a funcţiei, dacă a apărut o eroare etc.** De exemplu, dacă funcţia trebuie să întoracă un întreg pozitiv, atunci la situaţii de eroare să întoarcă valori negative, câte una pentru fiecare eroare. Dacă funcţia trebuie să întoarcă un pointer, atunci eroarea să fie semnalată prin pointerul NULL. De aceea, este secomandat **SA SE APELEZE:**

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

NU (aşa cum din comoditate apelează mulţi programatori) :

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

Evident forma a doua este mai scurtă, dar nu sunt tratate situaţiile de excepţie.

In acest context, **marea majoritate a funcţiilor standard C şi practic toate apelurile sistem Unix** întorc un rezultat care "spune" dacă funcţia/apelul s-a derulat normal sau dacă a apărut o situaţie deosebită. In caz de eşec funcţia / apelul sistem întoarce fie un întreg nenul (valoarea 0 este rezervată pentru succes), fie un pointer NULL etc.
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 situaţie de derulare anormală sistemul fixează o valoare nenulă ce indică cauza erorii**. Pentru detalii vezi `man errno`, precum şi 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 funcţii sistem *errno nu se seteaza la 0!* Pentru a se vedea şi în clar eroarea depistată se pot folosi funcţiile `strerror` şi `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); }
```

3.2. Principalele apeluri sistem Unix care operează cu procese

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

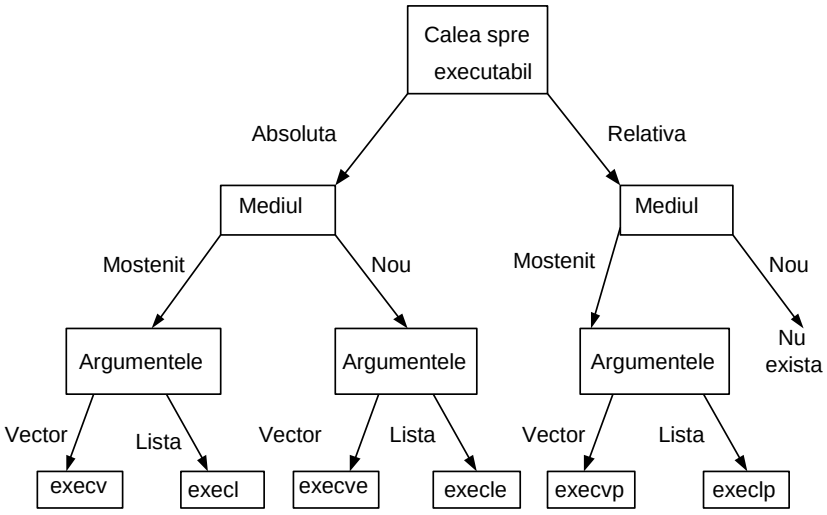
Funcţii specifice proceselor	
`fork()`	Duplică procesul curent; întoarce PID-ul fiu în părinte, 0 în fiu
`exit(n)`	Termină fiul cu codul de retur `n`
`wait(p)`	Aşteaptă terminare fiu cu primire codul de retur la `p`
`exec*(ne, ac)`	Lanseaza în procesul curent executabilul `ne` cu argumentele `ac`
`system(lcs)`	Execută ca shell comanda `lcs` şi întoarce codul de retur

Prototipurile acestor funcţii sunt descrise, de regula, in `<unistd.h>` Parametrii sunt:

- `n` este intreg – codul de retur cu care se termină procesul;
- `p` este un pointer la un întreg unde fiul întoarce codul de retur (extras cu funcţia `WEXITSTATUS);`
- `ne` este numele unui program executabil Unix;
- `ac` contine argumentele liniei de comandă: începe cu `ne` şi continuă cu argumentele liniei de comandă. Forma de livrare a `ac` este fie **o listă de stringuri** terminată cu un pointer NULL, fie un **tablou cu pointeri la stringuri** cu pointerul NULL pe ultima poziţie. Pentru argumentele din `ac` **nu** se tratează construcţiile de forma: ${ } ` ` * ? < > << >>. (Sarcina procesării acestora revine shell: aflarea valorii unei variabile de mediu, captarea ieşirii standard dintr-o construcţie între apostroafe inverse, specificări generice de fişiere, redirectarea fişierelor standard);
- `lcs` este o linie de comandă interpretabilă de către shell, deci se vor trata: ${ } ` ` * ? < > << >>

In caz de eşec, functiile întorc -1 si poziţionează `errno` se depisteaza ce eroare a aparut.

Tipurile de exec:



3.3. Exemple de lucrul cu procese

3.3.1. Utilizări simple fork exit, wait

Vom prezenta şi 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 situaţii, făcând o serie de modificări în această sursă.

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

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

Comentarea liniei 16: Părintele nu mai aşteaptă terminarea fiului şi acesta din urmă rămâne în starea zombie. Se tipăresc cele 21 de linii ca în primul caz. O observaţie interesantă: dacă ieşirea programului se redirectează într-un fişier pe disc, apar cele 21 linii. In schimb, dacă ieşirea se face direct pe terminal, apar doar liniile fiului. De ce oare? Rămâne un TO DO pentru studenţi.

Comentarea liniilor 12 şi 16: Se tipăresc 22 linii, cu aceeaşi observaţie de mai sus, de la comentarea liniei 16. Aici recomandăm modificări ale numărului liniilor tipărite de fiu (linia 10) şi 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 execuţiei acestui program? (Acoladele nu sunt necesare, dar le-am pus pentru a evidenţia mai bine corpul lui `if`). Să facem o primă analiză:

- Primul `fork` naşte un proces fiu. Ambele procese au de executat secvenţa: `if (fork()) fork(); printf("Salut\n");` Până acum avem **două** procese.
- Condiţia `fork` din `if` naşte un proces fiu pe alternativa false a lui `if`, căruia îi rămâne de făcut doar `printf("Salut\n");` Până acum avem **patru** procese.
- Aceeaşi condiţie `fork` din `if`, pe alternativa true (executată de părinte) îi rămâne de făcut `{fork();} printf("Salut\n");`.
- Fiecare `fork` dintre acolade mai naşte câte un proces fiu căruia îi mai rămâne de făcut `printf("Salut\n");` Avem încă două procese în plus, deci **şase** procese.
- In concluzie, avem şase (6) procese care au de executat `printf("Salut\n")`.

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

Pentru a aprofunda analiza, să rescriem puţin 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);
}
```

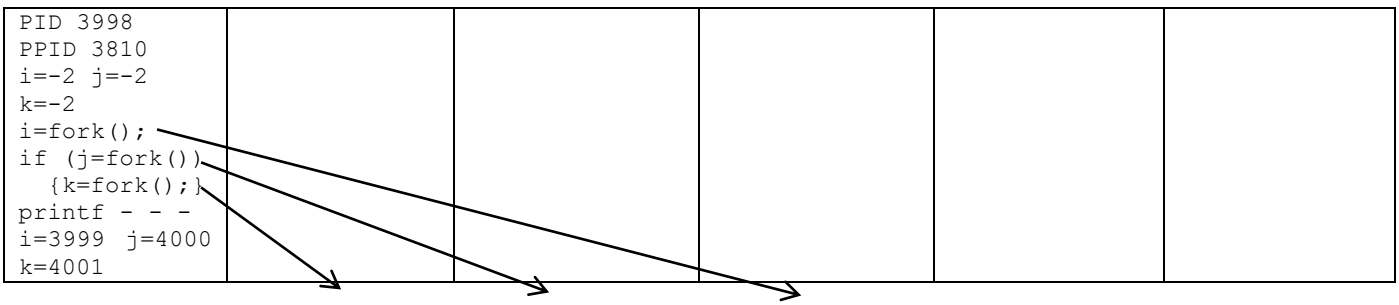
In fapt, am reţinut în variabilele `i, j, k` valorile PID-urilor create pe parcursul execuţiei. Rezultatul execuţiei este:

```
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 instrucţiuni:

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

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



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

In acest tabel, valoarea PPID este cea reală a părintelui creator, deşi la momentul terminării fiului părintele nu mai există, aşa că procesul intră în starea zombie. Vom reveni mai tarziu asupra evitării proceselor zombie.

3.3.2. Utilizări simple execl, execlp. execv, system

Urmatoarele două programe, desi diferite, au acelasi efect. Toate trei folosesc o comanda de tip `exec`, spre a lansa din ea comanda 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 pregateste 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 executa, pe rand, numai una dintre cele 5 linii, comentând pe celelalte 4. Ce se va intampla?

- Primul `execl` lanseaza `ls` prin cale absolută şi are acelaşi efect ca şi programul 1.
- Al doilea lansează `ls` prin directoarele din PATH, efectul este acelaşi.
- Al treilea cere `ls` pentru o listă de fişiere. Pentru cele care nu există, se dă mesajul: `/bin/ls: cannot access 'xx': No such file or directory` (in loc de xx apar numele fişierelor inexistente);
- Al patrulea exec va da mesajul: `/bin/ls: cannot access *.c: No such file or directory` **Nu este interpretat asa cum ne-am astepta!** De ce? Din cauza faptului ca specificarea *.c reprezinta o specificare generica de fisier, dar numai shell "stie" acest lucru si el (shell) inlocuieste

aceasta specificare, in cadrul uneia dintre etapele de tratare a liniei de comanda. La fel stau lucrurile cu evaluarea variabilelor de mediu, ${---}, inlocuirea dintre apostroafele inverse ` --- `, redirectarea I/O standard etc.

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

Funcţia `system` de fapt lansează mai întâi un shell, apoi în acesta lansează comanda `lcs`. Această lansare se poate face folosind un `execl`: **`execl("/bin/sh", "sh", "-c", lcs, NULL);`**
Doritorii pot să vadă sursa `system.c`, care este o funcţie simplă, de maximum 100 linii în care se includ comentariile, tratările cu `errno` ale posibilelor erori şi manevrarea unor semnale specifice. Sursa poate fi găsită la: `http://man7.org/tlpi/code/online/dist/procexec/system.c`

3.3.3. Un program care compileaza şi 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
```

Cefect, 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 şi 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 (strncmp("HOME", envp[i], 4)==0 || strncmp("LOGNAME", envp[i], 7)==0)
            printf("%s\n", envp[i]);
}
```

Secvenţa de execuţie 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$
```

3.3.4. Capitalizarea cuvintelor dintr-o listă de fişiere text

Se cere un program care primeşte la linia de comandă o listă de fişiere text. Se cere ca toate aceste fişiere să fie transformate în altele, cu acelaşi conţinut, dar în care fiecare cuvânt să înceapă cu literă mare. Se vor lansa procese paralele pentru prelucrarea simultană a tuturor fişierelor.

Pentru aceasta, vom crea mai întâi un program cu numele `cap` din sursa `cap.c`. Acesta primeşte la linia de comandă numele a două fişiere text, primul de intrare, al doilea de ieşire 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 == 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 fişier primit la linia de comandă şi 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`

3.3.5. 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 si positive. Se cere numărul de perechi care au suma un număr par, numărul de perechi ce au suma număr impar si numărul de perechi în care cel putin unul dintre argumente nu este număr strict pozitiv.

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

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

Parintele aşteaptă terminarea fiilor şi din codurile de retur întoarse de aceştia va afisa rezultatul cerut.

Vom da doua solutii:

1. Solutia 1 cu textul complet intr-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, nenumerice %d\n",pare, impare, nenum);
}
```

Solutia 2:

Se creaza programul `par.c` care primeşte la linia de comandă o pereche de argumente. Din această sursă se va consttiui 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 primeşte la linia de comandă n perechi de argumente. El va crea n procese fii şi în fiecare va lansa prin `exec` programul `par`. Din aceasta sursa se va consttiui 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, nenumerice %d\n",pare, impare, nenum);
}
```

Intrebare la ambele solutii: Ce se întamplă daca `wait` si `switch` nu sunt plasate în cicluri `for` succesive ci în acelaşi `for` care crează procesele fii?

3.4. Semnale Unix; exemple de utilizare

Un **semnal** este un anunţ asincron (nu se poate prevedea când se emite) primit de un proces sub forma unui număr întreg. procesul primitor nu ştie de unde vine, dar la primire are o acţiune implicită (terminarea procesului, ignorarea semnalului etc.) sau execută corpul unei funcţii. Prevenirea unui proces că ar putea primi un semnal se face prin apelul sistem **`signal`**. Un proces poate trimite un semnal altui proces prin apelul sistem **`kill`**, sau printr-o comandă **`kill`**.

3.4.1. Evitarea proceselor zombie

Pentru a evita starea acumularea de procese în starea zombie, **parintele trimite un semnal sistemului prin care să ignore aşteptrea după fii**:

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

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

In Linux efectul lui `signal` rămâne valabil până la un nou `signal`. In Unix BSD efectul unui `signal` este valabil o singură dată, şi atunci se procedează aşa:

```
#include <signal.h>
- - -
void waiter(){          // Functie de manipulare a apelurilor signal
   wait(0);           // Sterge fiul recent terminat
   signal(SIGCHLD, waiter); // Reinstalare handler signal
}// waiter
- - -
signal(SIGCHLD, waiter); // Plasat în partea de iniţializare
```

3.4.2. Schema client / server: adormire si deşteptare

Trecerea în adormire a unui proces se face trimiţându-i semnalul SIGSTOP, iar trezirea se face trimiţându-i semnalul SIGCONT.

In paradigma client / server programul ***server*** stă (este pus) în adormire şi va fi trezit de fiecare ***client*** ca să-i satisfacă o cerere, după care intră din nou în adormire.

Serverele sunt de două feluri: **servere iterative** şi **servere concurente**. La un server iterativ clienţii sunt serviţi unul după altul. La cele concurente se crează câte un fiu pentru fiecare cerere, iar serverul nu face `wait` pentru fii. La cele concurente apar multe procese zombie, motiv pentru care la iniţializare trebuie să se apeleze `signal(SIGCHLD, SIG_IGN)`.

Schematic, cele două variante de server apar sub forma:

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
main() {
```

```
    signal(SIGCHLD, SIG_IGN); // Numai la serverul concurent
    for ( ; ; ) {
        printf("%d doarme . . .\n", getpid());
        kill(getpid(), SIGSTOP); // Doarme, asteptand cereri
        // Daca PID este PID-ul serverului, un client il trezeste pentru cerere prin:
        // $ kill -SIGCONT PID sau kill(PID, SIGCONT);
```

Server iterativ	Server concurent

`printf("Servesc cererea . . .");`
`serveşteCerereClient( . . . );`
	`if (fork() == 0) {`
	`printf("Servesc cererea. . .");`
	`serveşteCerereClient( . . . );`
	`}`

```
    }
}
```

3.4.3. Aflarea unor informaţii de stare

Pentru un program care dureaza mult, se vrea din când în când să se afle stadiul calculelor:

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

// informatii globale de stare
   int  numar;

void  tipareste_stare(int semnal) {
// Tipareste informatiile de stare solicitate
   printf("Numar= %d\n", numar);
}//handlerul de semnal

main() {
   signal(SIGUSR1,tipareste_stare);
    //- - -
   for(numar=0; ; numar++) {

    // - - -

   }//for
}//main
```

Pentru tipărirea stadiului curent, cunoscând (ps) pidul programului, se dă comanda:

```
$ kill -SIGUSR1 pid
```

3.4.4. Tastarea unei linii în timp limitat

SE cere ca tastarea unei linii de la terminal să se facă în timp limitat (în cazul nostru 5 secunde), altfel se anulează citirea şi programul se aduce în starea dinaintea lansării citirii:

```
#include <stdio.h>
#include <setjmp.h>
#include <sys/signal.h>
#include <unistd.h>
#include <string.h>

   jmp_buf tampon;

   void handler_timeout (int semnal) {
      longjmp (tampon, 1);
```

```
}//handler_timeout

int t_gets (char *s, int t) {
   char *ret;
   signal (SIGALRM, handler_timeout);
   if (setjmp (tampon) != 0)
      return -2;
   alarm (t);
   ret = fgets (s, 100, stdin);
   alarm (0);
   if (ret == NULL)
      return -1;
   return strlen (s);
}//t_gets

main () {
   char s[100];
   int v;
   while (1) {
      printf ("Introduceti un string: ");
      v = t_gets (s, 5);
      switch (v) {
         case -1:
            printf("\nSfarsit de fisier\n");
            return(1);
         case -2:
            printf ("timeout!\n");
            break;
         default:
            printf("Sirul dat: %s a.Are %d caractere\n", s, v-1);
      }//switch
   }//while
}//t_gets.c
```

3.4.5. Blocarea tastaturii

Se blochează tastatura până când se tastează parola cu care s-a făcut login:

```
#define _XOPEN_SOURCE
#include <stdio.h>
#include <unistd.h>
#include<string.h>
#include <pwd.h>
#include <shadow.h>
#include <signal.h>
   main() {
      char *cpass, pass[15];
      struct passwd *pwd;
      struct spwd *shd;
      signal(SIGHUP, SIG_IGN);
      signal(SIGINT, SIG_IGN);
      signal(SIGQUIT, SIG_IGN);
      setpwent();
      pwd = getpwuid(getuid());
      endpwent();
      setspent();
      shd = getspnam(pwd->pw_name);
      endspent();
      setuid(getuid()); // Redevin userul real
      for ( ;; ) {
         strcpy(pass, getpass("...tty LOCKED!!"));
         cpass = crypt(pass, shd->sp_pwdp);
         if (!strcmp(cpass, shd->sp_pwdp))
            break;
```

```
        }
    }
//lockTTY.c
// Compilare: gcc -lcrypt -o lockTTY lockTTY.c
// User root: chown root.root lockTTY
// User root: chmod u+s lockTTY
//  Executie: ./lockTTY
```

3.5. Probleme propuse

1. Programul apelat compara doua sau mai multe numere primite ca argumente si returneaza cod 0 daca toate sunt egale, 1 altfel. Programul apelant citeste niste numere si spune daca sunt egale.

2. Programul apelat primeste ca argumente un nume de fisier si une sir de caractere si scrie in fisier sirul oglindit. Programul apelat citeste niste siruri de caractere si concateneaza oglindirile lor.

3. Programul apelat primeste ca argumente niste numere si returneaza cod 0 daca produsul lor este pozitiv, 1 daca e negativ si 2 daca e nul. Programul apelant citeste un sir de numere si afiseaza daca produsul lor este pozitiv, negativ sau zero.

4. Programul apelat primeste ca argumente doua numere naturale si un nume de fisier si scrie in fisier cel mai mic multiplu comun al numerelor. Programul apelant citeste un sir de numere naturale si afisieaza cel mai mic multiplu comun al lor.

5. Programul apelat primeste ca argumente doua numere naturale si un nume de fisier si scrie in fisier cel mai mare divizor comun al numerelor. Programul apelant citeste un sir de numere naturale si afisieaza cel mai mare divizor comun al lor.

6. Programul apelat primeste ca argumente doua numere si un nume de fisier si scrie in fisier produsul numerelor. Programul apelant citeste un sir de numere si afiseaza produsul lor.

7. Programul apelat primeste ca argumente trei nume de fisiere, primele doua continand cate un sir crescator de numere intregi, si scrie in al treilea fisier rezultatul interclasarii sirurilor din primele doua fisiere. Programul apelant citeste un sir de numere intregi, le sorteaza si scrie rezultatul sortarii.

8. Programul apelat primeste ca argumente un nume de fisier si niste siruri de caractere si le concateneaza, rezultatul fiind scris in fisierul dat ca prim argument. Programul apelat citeste niste siruri de caractere si le concateneaza.

9. Programul apelat primeste ca argumente niste numere si returneaza cod 0 daca suma lor este para si 1 altfel. Programul apelant citeste un sir de numere si afiseaza daca suma lor este para sau nu.

10. Programul apelat primeste ca argumente doua numere si returneaza cod 0 daca sunt prime intre ele si 1 altfel. Programul apelant citeste un sir de numere si determina daca sunt doua cate doua prime intre ele.

11. Programul apelat primeste ca argument un numar natural si returneaza cod 0 daca este prim si 1 altfel. Programul principal citeste un numar n si afiseaza numerele prime mai mici sau egale cu n.

12. Programul apelat primeste ca argumente doua nume de fisier si adauga continutul primului fisier la al doilea fisier. Programul apelant primeste un sir de nume de fisiere si concateneaza primele fisiere punand rezultatul in ultimul fisier.

13. Programul apelat primeste ca argumente doua numere si un nume de fisier si adauga in fisier toate numerele prime cuprinse intre cele doua numere date. Programul apelant citest un numar si scrie toate

numerele prime mai mici decat numarul dat, apeland celalalt program pentru intervale de cel mult 10 numere.

14. Programul apelat primeste ca argumente doua numere si un nume de fisier si scrie in fisier suma numerelor. Programul apelant citeste un sir de numere si afiseaza suma lor.

15. Programul apelat primeste ca argumente doua sau mai multe numere si returneaza cod 0 daca sunt doua cate doua prime intre ele, 1 altfel. Programul apelant citeste niste numere si spune daca sunt doua cate doua prime intre ele.