Sisteme de operare
Laborator 5
Controlul proceselor

1. Creati un nou subdirector *lab5/* in structura de directoare a laboratorului creata anterior (*SO/laborator*) si subdirectoarele aferente *doc src* si *bin*. Nu uitati sa actualizati variabila de mediu PATH pentru a include directorul *SO/laborator/lab5/bin*.

2. Scrieti un program C **fork-semantics.c** care declara o variabila intreaga *global* pe care o initializeaza cu valoarea 6 si o variabila globala de tip string *buf* initializata cu sirul de caractere "unbuffered write to stdout\n". Apoi, in functia *main*, declara o variabila intreaga *local* initializata cu valoarea 10.

Programul incepe prin a tipari variabila *buf* folosind apelul sistem *write*, iar apoi tipareste mesajul *"inainte de fork"* folosind de aceasta data functia de biblioteca *printf*. Apoi, programul foloseste apelul *fork* pentru a crea un nou proces si salveaza valoarea returnata in variabila *pid*. Procesul copil incrementeaza variabilele *global* si *local*, in vreme ce procesul parinte doarme pt 2 secunde folosind functia de biblioteca *sleep (man 3 sleep)*.

Ambele procese, parinte si copil, incheie cu acelasi cod tiparind valorile *pid*, *global* si *local*. Prin acelasi cod trebuie inteles ca ruleaza exact aceeasi linie de cod care apeleaza *printf*. Ce observati daca rulati programul interactiv? Dar daca rulati programul cu iesirea standard redirectata intr-un fisier, ca mai jos?

```
$ gcc -o fork-semantics fork-semantics.c
$ fork-semantics > out
```

Cum arata fisierul *out*? Care e diferenta fata de rularea interactiva?

3. Scrieti un program C **vfork-semantics.c** care modifica programul anterior apeland *vfork* in loc de *fork*. Mai exact, programul incepe prin a afisa pe ecran mesajul *"inainte de fork"* folosind functia de biblioteca *printf*. Apoi programul apeleaza *vfork* pentru a crea un nou proces si salveaza valoarea returnata in variabila *pid*. Procesul copil incrementeaza variabilele *global* si *local* ca in programul anterior dar nu mai afiseaza nimic pe ecran si apeleaza *exit(0)* imediat dupa modificarea variabilelor. Procesul parinte tipareste valorile *pid*, *global* si *local* pe ecran imediat dupa reintoarcerea din apelul *vfork* (nu mai apeleaza deloc *sleep*).

Ce observati daca rulati programul interactiv? Dar daca rulati programul cu iesirea standard redirectata intr-un fisier, ca mai jos?

```
$ gcc -o vfork-semantics vfork-semantics.c
$ vfork-semantics > out
```

Exista vreo diferenta fata de rularea interactiva?

4. Scrieti un program C **race.c** care defineste o functie *myprint* cu semnatura de mai jos:

```
void myprint(char *str);
```

Functia *myprint* tipareste pe ecran string-ul primit ca parametru caracter cu caracter, folosind apelul sistem *write* ca mai jos:

```
char c;
…
write(1, &c, 1);
```

Programul foloseste apelul sistem *fork* pentru a crea un nou proces. Dupa *fork*, procesul parinte apeleaza functia *myprint* folosind ca argument mesajul “This is the parent process printing\n”, si termina executia. La randul sau, procesul copil apeleaza functia *myprint* folosind ca argument mesajul “This is the child process printing\n”, si termina executia.

Rulati programul de mai multe ori. Ce se intampla cu cele doua mesaje afisate pe ecran? Cum va explicati ce se intampla?

Daca rularile de mai sus nu releva nimic important, modificati programul a.i. atat procesul parinte cat si procesul copil sa apeleze *myprint* intr-o bucla de 10 ori. Rulati programul si vedeti ce se intampla.