Sisteme de operare
Laborator 3
Controlul proceselor

1. Creati un nou subdirector *lab3/* 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/lab3/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 *sleep(2)*.

Ambele procese, parinte si copil, incheie cu acelasi cod tiparind valorile *pid*, *global* si *local*. 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*. De data aceasta procesul copil apeleaza *exit(0)* dupa modificarea variabilelor *global* si *local*, terminand executia fara a mai tipari nimic pe ecran. Procesul parinte nu mai apeleaza deloc *sleep*, marginindu-se ca imediat dupa reintoarcerea din apelul *vfork* sa tipareasca valorile *pid*, *global* si *local* pe ecran.

Obs: Pentru simplitatea output-ului, puteti renunta la variabila *buf* si la tiparirea valorii ei pe ecran.

4. Scrieti un program C **race.c** care foloseste apelul sistem *fork* pentru a crea un nou proces. Atat procesul parinte cat si copilul tiparesc un mesaj specific, sa zicem “this is the child/parent printing\n”, si termina executia. Tiparirea mesajului se face cate un caracter la un moment dat, folosind apelul sistem *write* ca mai jos:

```
char c;
…
write(1, &c, 1);
```

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

Indicatie: scrieti o functie *myprint* cu semnatura de mai jos care tipareste un string pe ecran, cate un caracter la un moment dat:

void myprint(char *str);

5. Scrieti un program C **my-exec-engine.c** care primeste ca parametru in linie de comanda un program executabil si apeleaza *fork* pentru a crea un nou proces. Procesul copil executa comanda primita ca argument folosind apelul sistem *execvp*. Parintele asteapta terminarea executiei copilului folosind apelul sistem *wait*.

6. Scrieti un program C **system.c** care simuleaza comportamentul programului anterior folosind functia de biblioteca *system*. Cum ati scrie varianta proprie a acestui program folosind *fork*/*exec*?

Indicatie: pentru a folosi functia *system* e nevoie sa concatenati string-urile furnizate ca parametri de apel ai programului (argv[i]). In acest sens, puteti folosi functia de biblioteca *strcat*.

7. Scrieti un program C **signal.c** care foloseste apelul fork pentru a crea un nou proces. Procesul parinte simuleaza executia unui task de lunga durata apeland functia de biblioteca *getchar*, si se va bloca in asteptarea unui caracter introdus de utilizator de la tastatura. Procesul copil tipareste pe ecran PID-ul sau si al parintelui sau si termina cu cod de stare 44.

Avand in vedere ca procesul parinte e blocat in asteptarea unui caracter, pentru a putea procesa evenimentul terminarii procesului copil asincron, parintele prinde semnalul SIGCHLD inregistrand un handler de semnal pt acest semnal inainte de a apela *fork*, cu ajutorul apelului sistem *signal*.

Handlerul de semnal SIGCHLD tipareste un mesaj care afiseaza numarul de semnal primit si cheama neblocant (cu flag-ul WNOHANG) apelul sistem waitpid pentru a afla PID-ul procesului copil care tocmai s-a terminat si afiseaza acest PID pe ecran.

La final, utilizatorul va debloca procesul parinte apasand pe o tasta iar procesul parinte, odata iesit din apelul *getchar*, se termina cu cod de stare 0.

8. Scrieti un program C **times.c** care masoara timpii de executie ai unuia dintre programele de mai sus folosind apelul sistem *times*.