# Sisteme de operare
# Laborator 10
# IPC System V, POSIX threads

**1.** Creati un nou subdirector *lab10/* 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/lab10/bin*.

**2.** Scrieti un program C **shm.c** care creeaza o zona de memorie partajata intre un proces parinte si un copil. Parintele creeaza zona de memorie partajata (puteti folosi IPC_PRIVATE) si o ataseaza la proces inainte de apelul *fork*. Procesul copil executa un loop cu 10 iteratii in care scrie succesiv, la fiecare iteratie, urmatorul mesaj in zona de memorie partajata:

"this is the child printing " + numarul iteratiei

Pentru a compune asemenea mesaje puteti folosi functia de biblioteca *sprintf*. Mesajele din fiecare iteratie se adauga (*append*) la sfarsitul mesajului scris in iteratia anterioara (cu exceptia primei iteratii).

Procesul parinte executa si el un loop cu 10 iteratii in care, la fiecare iteratie, citeste unul dintre mesajele scrise de procesul copil si le afiseaza pe ecran.

Dezvoltati o solutie folosind semafoare System V sau semafoare POSIX cu nume care sa permita coordonarea (sincronizarea) accesului celor doua procese la zona de memorie partajata a.i. executia programului sa fie cea anticipata, si anume dupa fiecare mesaj scris de copil in memoria partajata parintele il citeste si il scrie pe ecran. Solutia trebuie sa fie strans sincronizata, in sensul in care copilul scrie un mesaj si asteapta ca parintele sa-l citeasca si abia apoi scrie urmatorul mesaj.

*Obs*: *ambele procese executa un loop cu 10 iteratii dar nu puteti face nici o presupunere asupra vitezei relative de executie a celor doua procese*.

La final, apelati *shmctl* si *semctl* cu IPC_RMID pentru a sterge memoria partajata si semafoarele din sistem (eventual *shm_unlink*, daca folositi semafoare POSIX cu nume).

**3.** Scrieti doua programe C **msg-echod.c** si **msg-echo.c** care implementeaza un serviciu de *echo* la fel ca in laboratorul 8, de data aceasta folosind cozi de mesaje System V IPC in loc de FIFOs. Practic, in loc sa foloseasca un fisier *FIFO*, programul **msg-echod.c** creeaza o coada de mesaje cu ajutorul *msgget* (parametrul *key* il generati cu functia *ftok*). Apoi apeleaza in bucla functia *msgrcv* pentru a primi mesaje de la clienti pe care le scrie inapoi in coada de mesaje (serviciu de *echo*) folosind *msgsnd*. Programul **msg-echod.c** se lanseaza in background ca serviciu de echo dupa cum urmeaza:

```
$ gcc -o msg-echod  msg-echod.c
$ msg-echod keyfile &              # keyfile e un fisier pe care il creati voi
$ ipcs -q                          # vizualizati coada creata
```

Programul **msg-echo.c** foloseste parametrul primit in linie de comanda (*keyfile*) ca argument pt functia *ftok* si astfel poate identifica si obtine cu ajutorul *msgget* un ID pentru coada de mesaje creata de *echod*. Apoi, citeste in bucla caractere de la tastatura folosind functia *fgets* si asambleaza mesaje cu tipul dat de PID-ul sau (obtinut cu *getpid*) pe care le trimite in coada de mesaje cu ajutorul functiei *msgsnd*. Apoi citeste din coada de mesaje cu ajutorul *msgrcv* ecoul caracterelor

trimise catre serviciul *echod* pe care, odata primite, le tipareste pe ecran. Programul se termina cand utilizatorul apasa Ctrl-c.

```
$ gcc -o msg-echo msg-echo.c
$ msg-echo keyfile
```

La final, nu uitati sa faceti curatenie:

```
$ jobs
$ kill %<n>          # n e numarul de job al msg-echod afisat
                     # de comanda jobs de mai sus
$ ipcs -q
$ ipcrm -q <id>      # id e id-ul cozii de mesaje tiparit de comanda
                     # ipcs de mai sus
```

Pentru a evita ultimele doua comenzi de mai sus, in progamul **msg-echod.c** puteti sa prindeti semnalul SIGTERM si sa stergeti coada de mesaje din program atunci cand se executa comanda *kill* de mai sus.

**4.** Modificati programul C **msg-echod.c** de mai sus pentru a oferi un serviciu ***multi-threaded*** de *echo* folosind cozi de mesaje System V IPC. Noul program, **msg-mt-echod.c**, creeaza o coada de mesaje cu ajutorul *msgget* (parametrul *key* il generati cu functia *ftok*). Apoi apeleaza in bucla functia *msgrcv* pentru a primi mesaje de la clienti, Spre deosebire de **msg-echod.c**, care este *single-threaded*, **msg-mt-echod.c** creeaza un thread POSIX separat pentru fiecare nou mesaj primit. Serverul copiaza mesajul primit cu *msgrcv* si il paseaza noului thread creat care il scrie inapoi in coada de mesaje (serviciu de *echo*) folosind *msgsnd*. Programul **msg-mt-echod.c** se lanseaza in background ca serviciu de *echo* dupa cum urmeaza:

```
$ gcc -o msg-mt-echod  msg-mt-echod.c
$ msg-mt-echod keyfile &        # keyfile e un fisier pe care il creati voi
$ ipcs -q                       # vizualizati coada creata
```

Pentru a testa functionalitatea serverului folositi programul client **msg-echo.c** dezvoltat in laboratoarele trecute.

La final ca la exercitiul anterior, nu uitati sa faceti curatenie in sistem stergand coada de mesaje fie prin program, fie din shell.

**5.** Rescrieti programul C **race.c** din laboratorul 5 in care in loc sa folositi doua procese, parinte si copil, care scriu concurent, caracter cu caracter, propriul string pe ecran, veti folosi doua thread-uri POSIX. Pentru a evidentia mai clar race condition-ul, fiecare thread repeta operatia de tiparire pe ecran de un numar de ori, in bucla. Folositi un *semafor* sau un *mutex* POSIX pentru a impiedica producerea race condition-ului si a obtine un output neintretesut pe ecran. La final, daca ati folosit un semafor, nu uitati sa-l stergeti din sistem.

*Obs*: Avand in vedere ca programul foloseste thread-uri, puteti folosi semafoare POSIX fara nume direct, apeland *sem_init* cu parametrul corespunzator.

**6.** Scrieti un program C **prod-cons.c** care implementeaza o solutie a problemei producator-consumator folosind doua thread-uri POSIX, unul pentru producator si unul pentru consumator. Pentru implementarea sincronizarii folositi monitoare (mai exact, fiind vorba de POSIX, un mutex si doua variabile conditie, una pentru buffer plin si alta pentru buffer gol).

Care este diferenta in termeni de complexitate a programarii intre solutia cu thread-uri si cea cu semafoare si memorie partajata System V  din laboratorul trecut?