```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/msg.h>
#include <sys/wait.h>

struct message
{
       long tip;
    char buffer[50];
    int size;
};

/* Programul primeste doua argumente, un nume de fisier din directorul curent de lucru si o cale catre un director. Programul
   copiaza fisierul transmis ca prim argument in directorul transmis ca al doilea argument sub acelasi nume.

   Pentru a face acest lucru, procesul tata creeaza un proces fiu care face chdir() in directorul destinatie. Procesul tata trimite
   continutul fisierului destinatie la procesul fiu pentru a il scrie in destinatie. Comunicarea intre ei se face prin o coada de mesage.
   Continutul fisierul este transmis in bucati de cate 50 de octeti maxim. IPC ul este de tip privat, deci este accesibil numai procesului creator
   si a descendentilor acestuia. Acesta trebuie sters dupa ce nu mai este folosit

*/
int main(int argc, char *argv[]) {

    if (argc < 3) {
       puts("Numar insuficient de argumente");
       return -1;
    }


    //Am creat un IPC privat cu drepturi de citire si scriere pentru propietar
       int msgqid = msgget(IPC_PRIVATE, S_IRUSR | S_IWUSR );

    int child_pid;

    if ((child_pid = fork())) {
        //instructiuni executate doar de procesul tata

        int descriptor = open(argv[1], O_RDONLY);

        //tipul mesajului este pid ul procesului destinatar
        struct message message_for_child;
        message_for_child.tip = child_pid;

        //citim din fisier si trimitem continutul in coada de mesaje
        while((message_for_child.size = read(descriptor, message_for_child.buffer, 50)) > 0) {
             msgsnd(msgqid, &message_for_child, sizeof(message_for_child), 0);
        }

        //dupa ce am terminat de parcurs fisierul ii trimitem fiului un mesaj de dimensiune 0 pentru ca acesta sa stie
```

```
        //ca a terminat de primit continutul fisierului si nu ar mai trebui sa astepte
dupa alte mesaje
        message_for_child.size = 0;
        msgsnd(msgqid, &message_for_child, sizeof(message_for_child), 0);

        close(descriptor);

        //asteptam fiul sa isi termine executia
        wait(NULL);

    } else {

        chdir(argv[2]);

        int descriptor = open(argv[1], O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
        struct message message_from_father;

        //fiul citeste din coada de mesaje doar mesajele care ii sunt destinate lui
(adica cele care au ca "tip" pid ul sau)
        msgrcv(msgqid, &message_from_father, sizeof(message_from_father),
getpid()/*aici specificam tipul mesajului asteptat*/, 0);

        //citim bucati din fisier atata timp cat nu au dimensiunea 0
        while (message_from_father.size > 0) {
            write(descriptor, message_from_father.buffer, message_from_father.size);
            msgrcv(msgqid, &message_from_father, sizeof(message_from_father),
getpid(), 0);
        }

        close(descriptor);

        //dupa ce am terminat de folosit coada, o stergem. este datoria fiului sa
stearga coada deoarece el este ultimului care
        //actioneaza asupra sa. daca am sterge coada din interiorul procesului tata
este posibil ca instructiunile de trimitere
        //al ultimului mesaj si cea de a sterge coada sa se execute inainte ca fiul sa
apuce sa citeasca ultimul mesaj. Astfel,
        //o sa ne aflam intr o stare de inconsistenta
        msgctl(msgqid, IPC_RMID, NULL);

    }

       return 0;
}



#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/msg.h>

struct message
{
       long tip;
    int zi;
    int luna;
    int an;
};
```

```c
//Program care trimite un mesaj intr o coada de mesaje a carui cheie este data ca
argument in linia de comanda
int main(int argc, char *argv[]) {


    if (argc < 2) {
       puts("Numar insuficient de argumente");
       return -1;
    }

    long key;
    //Pentru a citi cheia de mesaje si a o stoca intr o variabila de tip long folosim
functia sscanf
    //deoarece atoi() functioneaza pe reprezentare in baza 10 a numerelor
    sscanf(argv[1], "%li", &key);

       int msgqid = msgget(key, 0);

    struct message mymessage;

    mymessage.tip = 1;
    mymessage.zi = 1;
    mymessage.luna = 2;
    mymessage.an = 2018;

    //apelul sistem de trimitere a unui mesaj in coada
    msgsnd(msgqid, &mymessage, sizeof(mymessage), 0);

       return 0;
}



#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/msg.h>

struct message
{
       long tip;
    int zi;
    int luna;
    int an;
};

//Program care citeste un mesaj dintr o coada de mesaje a carui cheie este data ca
argument in linia de comanda
//Programul o sa fie compilat cu optiunea -fno-stack-protector
int main(int argc, char *argv[]) {

    if (argc < 2){
        puts("Numar insuficiente de argumente");
        return -1;
    }

    long key;
```

```
    //Retineti modul prin care convertim cheia intr o variabila de tip long
    sscanf(argv[1], "%li", &key);

    int msgqid = msgget(key, 0);

    struct message mymessage;

    msgrcv(msgqid, &mymessage, sizeof(struct message), 1, 0);

    printf("Zi: %d Luna: %d An:%d\n", mymessage.zi, mymessage.luna, mymessage.an);

       return 0;
}



#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <ctype.h>

/*
  Programul primeste in linia de comanda calea catre doua fisiere, un fisier de input
si un fisier de output. Programul
  copiaza din fisierul de input tot continutul in fisierul de output transformand
fiecare litera in uppercase.

  Pentru a face acest lucru procesul tata lanseaza un proces fiu care transforma
literele mici in litere mari. Comunicarea dintre
  tata si fiu se face prin intermediul tuburilor. Se foloseste un tub pentru a
transmite informatie dinspre tata catre fiu si un tub
  pentru a transmite informatii dinspre fiu catre tata. Se verifica posibilele cazuri
de eroare ce pot aparea.
*/

int main(int argc, char *argv[]) {

  if (argc < 3){
    puts("Numar insuficiente de argumente");
    return -1;
  }

  //vectorii cu descriptorii catre cele doua tuburi
  int pipe_from_father_to_child[2];
  int pipe_from_child_to_father[2];

  //apelul sistem pipe() prin care cerem kernel-ului sa ne creeze mecanismele
  pipe(pipe_from_father_to_child);
  pipe(pipe_from_child_to_father);

  if (fork()){
    //instructiuni specifice procesului tata

    /*La apelul fork() descriptorii tatalui sunt mosteniti de catre procesul fiu
astfel ca acum avem doi descriptori
      pentru scriere in pipe_from_father_to_child (unul in tata si unul in fiu) si doi
de citire (unul in tata si unul in fiu),
```

```
      la fel s a intamplat si pentru tubul pipe_from_child_to_father. Cum tatal
foloseste pipe_from_father_to_child doar sa scrie
      informatia catre fiu si pipe_from_child_to_father doar sa citeasca informatia de
la fiu nu are nevoie de descriptorul de citire
      catre primul tub si de cel de scriere pentru al doilea. Le inchidem ca sa nu
avem problema de sincronizare.
    */
    close(pipe_from_father_to_child[0]);
    close(pipe_from_child_to_father[1]);

    int descriptor_input_file = open(argv[1], O_RDONLY);

    //verificam daca fisierul de input s a deschis cu succes
    if (descriptor_input_file < 0){
      puts("Nu se poate deschide fisierul de input");

      //inchidem si ceilalti descriptori ai tatalui
      close(pipe_from_father_to_child[1]);
      close(pipe_from_child_to_father[0]);

      //asteptam ca procesul fiu sa isi termine executia nu vrem sa retinem vreo
informatie despre procesul fiu astfel ca
      //transmitem pointerul NULL catre apelul sistem wait()
      wait(NULL);
      return -1;
    }

    int descriptor_ouput_file = open(argv[2], O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);

    char buffer[50];
    int numer_of_bytes_read_from_input_file;

    //citim cat timp avem informatie in fisier
    while ((numer_of_bytes_read_from_input_file = read(descriptor_input_file, buffer,
50)) > 0) {
      //transmitem fiecare bucata de informatie cititat din fisier catre fiu prin
tubul aferent
      //apelul va fi blocant doar daca tubul devine plin si inca exista un descriptor
de citire catre acest pipe (vezi laborator 6)
      write(pipe_from_father_to_child[1], buffer,
numer_of_bytes_read_from_input_file);

      //citim informatia procesata venita de la fiu prin cel de al doilea tub
      //apelul va fi blocant daca nu exista informatie disponibila in fisier
      read(pipe_from_child_to_father[0], buffer, numer_of_bytes_read_from_input_file);

      //scriem in fisierul de output informatia
      write(descriptor_ouput_file, buffer, numer_of_bytes_read_from_input_file);
    }

    //inchidem toti descriptorii folositi de procesul tata (cele doua catre fisiere si
cele doua catre tuburi)
    close(pipe_from_father_to_child[1]);
    close(pipe_from_child_to_father[0]);
    close(descriptor_input_file);
    close(descriptor_ouput_file);

    //asteptam ca procesul fiu sa isi termine executia
    wait(NULL);

  } else {
    //instructiuni specifice procesului fiu
```

```
    /* Aceasi explicatie ca mai sus, fiul nu are nevoie sa scrie in
pipe_from_father_to_child si nu are nevoie sa citeasca din cel
       de al doilea tub. Astfel, inchidem cei doi descriptori inainte sa facem alte
operatii.
    */
    close(pipe_from_father_to_child[1]);
    close(pipe_from_child_to_father[0]);

    char buffer[50];
    int numer_of_bytes_read_from_pipe;

    //Citirea din tub este blocanta daca nu exista informatii disponibile. Procesul
fiu va intra in asteptare pana cand o sa fie
    //octeti disponibili. In momentul in care tatal inchide descriptorul de scriere in
pipe_from_father_to_child, nu o sa mai existe
    //scriitori pentru acest tub deoarece si fiul l a inchis pe al lui anterior,
astfel ca apelul read() va intoarce end-of-file si se
    //va iesi din bucla while(). Daca nu am inchide in procesul tata descriptorul de
scriere se va creea o situatie de deadlock
    //deoarece tatal o sa fie blocat la wait() pe un fiu care este si el blocat la
randul lui de un apel la read()
    while((numer_of_bytes_read_from_pipe = read(pipe_from_father_to_child[0], buffer,
50)) > 0) {

      //logica efectiva de transformare a literelor
      for (int i = 0; i < numer_of_bytes_read_from_pipe; ++i) {
         if (isalpha(buffer[i]) && islower(buffer[i])){
           buffer[i] = toupper(buffer[i]);
         }
      }

      //scriem informatia procesata in tubul catre tata. regulile de scriere intr un
tub le am discutat mai sus
      write(pipe_from_child_to_father[1], buffer, numer_of_bytes_read_from_pipe);
    }

    //inchidem ceilalti descripotri folositi de procesul fiu
    close(pipe_from_father_to_child[0]);
    close(pipe_from_child_to_father[1]);

  }

  //aici o sa ajunga si procesul tata si procesul fiu
  return 0;
}



#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <ctype.h>


/* Programul numara cate linii se afla in fisierul transmis ca parametru. Pentru a
face acest lucru el se foloseste de utilitarele
```

```
   din linia de comanda cat si wc. "cat" afiseaza continutul unui fisier la standard
output iar "wc" citeste de la standard input si
   afiseaza cate cuvinte citeste, cu argumentul "-l" el numara cate linii citeste.
Echivalentul in linia de comanda este:

   $ cat fisier | wc -l

   Astfel, o sa avem doua tuburi. Un tub prin care comunica procesul lansat dupa
imaginea lui "cat" cu procesul lansat dupa imaginea
   lui "wc", si un tub prin care comunica procesul "wc" cu procesul tata. Redirectam
iesirea standard a lui cat si intrarea standard
   al lui wc in primul tub, iar iesirea standard a lui wc si intrarea standard al
tatalui in cel de al doilea tub

*/
int main(int argc, char *argv[]) {

  if (argc < 2){
    puts("Numar insuficiente de argumente");
    return -1;
  }

  //cele doua tuburi
  int pipe_from_cat_to_wc[2];
  int pipe_from_wc_to_father[2];

  //apelul sistem pentru a creea tubul prin care o sa comunice procesele cat si wc
  pipe(pipe_from_cat_to_wc);

  if (fork() == 0) {
     //inchidem descriptorul de citire din primul tub deoarece "cat" nu are nevoie de
el
     close(pipe_from_cat_to_wc[0]);

     //redirectam iesirea standard al acestui proces fiu in tubul nostru
     close(STDOUT_FILENO);
     dup(pipe_from_cat_to_wc[1]);
     close(pipe_from_cat_to_wc[1]);

     //inlocuim imaginea procesului cu imaginea lui "cat" cu fisierul transmis ca
parametru
     execlp("cat", "cat", argv[1], NULL);
  }

  //apelul sistem pentru a creea tubul prin care o sa comunice procesul "wc" cu tatal
  pipe(pipe_from_wc_to_father);

  if (fork() == 0) {

     //inchidem descriptorul de scriere in primul tub deoarece "wc" nu are nevoie de
el
     close(pipe_from_cat_to_wc[1]);

     //redirectam intrarea standar al lui "wc" in primul tub
     close(STDIN_FILENO);
     dup(pipe_from_cat_to_wc[0]);
     close(pipe_from_cat_to_wc[0]);

    //inchidem descriptorul de citire din cel de al doilea tub deoarece nu este nevoie
de el
     close(pipe_from_wc_to_father[0]);
```

```
     //redirectam iesirea standard in cel de al doilea tub, prin care se face
comunicare cu tatal
     close(STDOUT_FILENO);
     dup(pipe_from_wc_to_father[1]);
     close(pipe_from_wc_to_father[1]);

    //inlocuim imaginea acestui proces fiu cu imaginea lui wc
     execlp("wc", "wc", "-l", NULL);

  }

  //inchidem in toti descriptorii catre primul tub din procesul tata deoarece nu sunt
folositi de catre el
  close(pipe_from_cat_to_wc[0]);
  close(pipe_from_cat_to_wc[1]);

  //inchidem descriptorul de scriere in cel de al doilea tub
  close(pipe_from_wc_to_father[1]);

  //redirectam intrarea standard a tatalui in tubul prin care comunica cu fiul lansat
dupa imaginea lui "wc"
  close(STDIN_FILENO);
  dup(pipe_from_wc_to_father[0]);
  close(pipe_from_wc_to_father[0]);

  //citim rezultatul procesarii
  int numer_of_lines;
  scanf("%d", &numer_of_lines);

  printf("Numarul de linii din fisierul %s este %d\n",  argv[1], numer_of_lines);

  return 0;
}



#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/shm.h>
#include<sys/sem.h>
#include<sys/wait.h>
#include<sys/stat.h>

void process(char* value) {
    //functia de procesare
    printf("Proccesing %s.....\n", value);
    //Simulam o procesare indelungata
    sleep(30);
}

/* Programul exemplifica conceptul de pooling de procese. Descrierea detaliata a fost
predata in cadrul laboratorului.

*/
int main(int argc, char* argv[]){


    int shmid = shmget( IPC_PRIVATE, 50, S_IRUSR | S_IWUSR );

    int semafor_slave = semget( IPC_PRIVATE, 1, S_IRUSR | S_IWUSR );
    int semafor_master = semget( IPC_PRIVATE, 1, S_IRUSR | S_IWUSR );
```

```
    semctl(semafor_slave,0,SETVAL,0);
    semctl(semafor_master,0,SETVAL,0);

    for (int i = 0; i <  7; ++i) {
        if (fork() == 0) {

            //Slaves
            char* shared_memory = shmat( shmid, NULL, SHM_RND );

            struct sembuf wait_for_data;
            wait_for_data.sem_num = 0;
            wait_for_data.sem_op = -1;
            wait_for_data.sem_flg = 0;

            struct sembuf notify_master;
            notify_master.sem_num = 0;
            notify_master.sem_op = 1;
            notify_master.sem_flg = 0;

            while(1) {
                 char value[50];

                 semop(semafor_slave,&wait_for_data,1);
                 sscanf(shared_memory, "%s", value);
                 semop(semafor_master,&notify_master,1);
                 process(value);

            }

        }
    }

    //Master
    struct sembuf notify_slaves;
    notify_slaves.sem_num = 0;
    notify_slaves.sem_op = 1;
    notify_slaves.sem_flg = 0;

    struct sembuf wait_for_slaves;
    wait_for_slaves.sem_num = 0;
    wait_for_slaves.sem_op = -1;
    wait_for_slaves.sem_flg = 0;

    char* shared_memory = shmat( shmid, NULL, SHM_RND );

    while(1) {
        char value[50];
        printf("Please input data:");
        scanf("%s", value);

        sprintf(shared_memory, "%s", value);

        semop(semafor_slave, &notify_slaves, 1);
        semop(semafor_master, &wait_for_slaves, 1);
    }

    return 0;
}



#include <stdlib.h>
```

```c
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>


/* Programul se foloseste de programul count-lines pe care il gasiti in arhiva pentru
a numara toate liniile din toate fisierele
   transmise ca parametru. Ca sa realizeze acest lucru se foloseste de o strategie de
tip "fork-join" in care lanseaza un proces
   pentru fiecare fisier in parte. Dupa aceasta asteapta toti fii sa isi termine
executia pentru ca la final sa faca agregarea
   rezultatelor.
*/
int main(int argc, char *argv[]) {

  //verific daca am argumente
  if (argc < 2){
    puts("No arguments");
    return -1;
  }

  //lansez toate procesele fiu
  for (int i = 1; i < argc; i++) {
      if (fork() == 0){
        //instructiuni executate de procesul fiu
        //Retineti diferenta dintre execl, execlp, execv, execvp
        execl("./count-lines", "count-lines", argv[i], NULL);
      }
  }

  int aggregate = 0;
  int status;

  //asteptam toate procesele fiu
  while(wait(&status) != -1) {
    if (WIFEXITED(status)){
      aggregate += WEXITSTATUS(status);
    }
  }

  printf("Numarul de total de linii din fisiere este:%d\n", aggregate);

  return 0;
}



#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>


/* Programul numara liniile dintr un fisier transmis ca parametru si returneaza in
codul de retur valoarea numaratoarei.
```

```
   Se verifica posibilele cazuri de eroare (numar insuficient de argumente, daca
fisierul poate sa fie deschis) si returneaza 0
   in acest caz.

   Se compileaza cu :
   $ gcc count-lines.c -o count-lines
*/
int main(int argc, char *argv[]) {

    if (argc < 2){
        return 0;
    }

    int descriptor = open(argv[1], O_RDONLY);

    if (descriptor < 0){
        return 0;
    }

    int line_count = 0;
    char ch;

    while (read(descriptor, &ch, 1) == 1){
        if (ch == '\n') {
            line_count++;
        }
    }

       return line_count;
}



#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/msg.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/ipc.h>

union semun {
    int              val;
    struct semid_ds *buf;
    unsigned short  *array;
    struct seminfo  *__buf;
};

//Programul exemplifica folosirea apelurilor sistem ce citesc informatii despre
diferitele tipuri de IPC uri
int main(int argc, char *argv[]) {


    if (argc < 4) {
       puts("Numar insuficient de argumente");
       return -1;
    }

    long msg_key;
```

```
    long shm_key;
    long sem_key;

    //citim cheile externe ale ipc urilor
    sscanf(argv[1], "%li", &msg_key);
    sscanf(argv[2], "%li", &shm_key);
    sscanf(argv[3], "%li", &sem_key);

    int msgqid = msgget(msg_key, 0);
    int shmid = shmget(shm_key, 0/*numarul de octeti este luat in considerare doar
daca aceasta operatie creeaza un IPC nou*/, 0);
    int semid = semget(sem_key, 0/*numarul de semafoare din vector este luat in
considerare doar daca aceasta operatie creeaza un IPC nou */,
                       0);

    //structurile ce incapsuleaza informatiile despre fiecare tip de IPC in parte
    struct msqid_ds info_msg;
    struct shmid_ds info_shm;
    struct semid_ds info_sem;

    union semun semopts;
    //campul buf este un pointer, deci trebuie sa indice catre o zona de memorie
alocata in prealabil
    semopts.buf = &info_sem;

    msgctl(msgqid, IPC_STAT, &info_msg);
    shmctl(shmid, IPC_STAT, &info_shm);

    //apelul semctl are o definitie mai complicata decat celelalte doua, si necesita
definirea tipului de date union in programul nostru
    //cum am facut mai sus
    semctl(semid, 0, IPC_STAT, semopts);

    printf("Numarul de mesage din coada este %d si numarul de octeti maxim este %d\n",
(int) info_msg.msg_qnum, (int) info_msg.msg_qbytes);
    printf("Dimensiunea in octeti a zonei de memorie este %d iar pid ul procesului
creator este %d\n", (int) info_shm.shm_segsz, (int) info_shm.shm_cpid );
    printf("Numarul de semafoare din vectorul de semafoare este %d\n", (int)
info_sem.sem_nsems);

      return 0;
}
```