```
#include <iostream>
#include<string.h>
#include<unistd.h>
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<errno.h>
#include<sys/wait.h>
#include<pthread.h>
#include<semaphore.h>

using namespace std;
//lab2_1.c
int main() {
   char * buf="Hello world\n";
   //1 este descriptorul rezervta pentru afisarea pe ecran
   write(1, buf, strlen(buf));
   return 0;
}
//lab2_2.c
struct stat sb;
int main(int argc, char *argv[])
{
   //informatii suplimentare folosind structura stat fara a deschide fisierul
   //argv[0] este numele executabilului
   if(stat(argv[1], &sb))
   {
      perror(argv[1]);//afisare mesaj de eroare
      return errno; //variabila globala care retine automat codul de eroare
   }
   //%jd converteste un off_set pentru afisare
   printf("Fisierul %s socupa %jd bytes pe disk\n", argv[1], sb.st_size);

   //deschidere fisier fd=file descriptor
   int fd=open(argv[1], O_RDONLY, S_IRUSR |S_IWUSR);
   if(fd==-1)
   {
      perror("nu s-a putut deschide fisierul");
      return errno;
   }
```

```
    //citire din fisier in buffer
    char buf[100];
    read(fd, buf, 99);
    buf[100]='\0';

    //inchidere fisier din care am citit
    int ret=close(fd);
    if(ret==-1)
    {
        perror("eroare la inchiderea fisierului");
        return errno;
    }
    //deschidere fisier de scris
    fd=open(argv[2], O_CREAT| O_WRONLY, S_IRUSR |S_IWUSR);
    if(fd==-1)
    {
        perror("nu s-a putu deschide fisierul");
        return errno;
    }
    //scriere in fisier
    write(fd, buf, strlen(buf));
    //inchidere fisier
    ret=close(fd);
    if(ret==-1)
    {
        perror("eroare la inchiderea fisierului ");
        return errno;
    }
    return 0;
}
//lab4_1.c
int main()
{
    pid_t pid=fork();
    if(pid<0)
        //errno variabila globala care retine codul pentru eroare
        return errno;
    else
        if(pid==0)
        {
            //instructiuni copil
```

```
            char *argv[]={"ls", NULL};//are un argument
            //functia ls este definita in /bin
            //argv= argumentele programului
            //env=NULL argumentele sistemului
            execve("/bin/ls", argv, NULL);
            perror(NULL);

        } else
        {
            //instructiuni parinte
            printf("My ID=%d, Child PID=%d\n", getppid(), getpid());
            //parintele primeste id-ul copilului
            //fiind in parinte id-ul copilului va fi id-ul curent
            //id-ul parintelui va fi in parintele parintelui
            wait(NULL);//asteapta sa termine copilul ca sa nu ramana copilul orfan
                    //si sa ajunga la bunici
             printf("Child finished\n", getpid());

        }
        return 0;
}
//lab4_2.c
int main(int argc, char *argv[])
{
    int n=atoi(argv[1]);//transformare char in int

    pid_t pid=fork();
    if(pid<0)
        return errno;
    else
        if(pid==0)
        {
            //copil
            printf("%d: ", n);
            while(n>1)
            {
                printf("%d ",n);
                if(n%2==0)
                    n=n/2;
                else
                    n=3*n+1;
```

```
            }
            printf(" 1\n");
        } else
        {
            //parinte
            wait(NULL);//asteapta dupa copil
                    //returneaza pid-ul copilului
            printf("Child finished\n", getpid());
        }
        return 0;
}
//lab4_3.c
int main(int argc, char *argv[])
{
    printf("Starting parent %d\n", getpid());
    for(int i=1;i<argc;i++)
    {
        pid_t pid = fork();
        if (pid < 0)
            return errno;
        else if (pid == 0) {
            //copil
            int n=atoi(argv[i]);
            printf("%d: ", n);
            while (n > 1) {
                printf("%d ", n);
                if (n % 2 == 0)
                    n = n / 2;
                else
                    n = 3 * n + 1;
            }
            printf(" 1\n");
            printf("Done parent %d Me %d\n", getppid(), getpid());
        }
    }
    for(int i=1;i<argc;i++)
        wait(NULL);
    printf("Done Parent %d Me %d\n", getppid(), getpid());
    return 0;
}
//lab5.c
```

```
int main(int argc, char *argv[])
{
   printf("Starting parent %d\n", getpid());

   //creare memorie partajata
   int shm_fd=open("myshm", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
   //stabilire dimensiune memorie -> multiplu de dimensiunea paginilor 4096
   size_t n=(argc-1)*getpagesize();
   //dimensionare memorie
   if(ftruncate(shm_fd, n)==-1)
   {
      perror(NULL);
      //diustrugere memorie
      shm_unlink("myshm");
      return errno;
   }
   //incarcare in spatiul procesului
   //mmap intoarce un pointer de tip void => facem conversie explicita
   //0 ca sa decida kernelul ce adresa pune
   //PROT_READ-> drepturi de acces
   int *p=(int *) mmap(0, n, PROT_READ, MAP_SHARED, shm_fd, 0);
   if(p==MAP_FAILED)
   {
      perror(NULL);
      shm_unlink("myshm");
      return errno;
   }
   for(int i=1;i<argc, i++)
   {

      int x = atoi(argv[i]);
      pid_t pid = fork();
      if (pid < 0)
         return errno;
      else if (pid == 0)
      {
         //copil
         //aloc cate o felie din memorie
         int *vect = (int *) mmap(0, n, PROT_WRITE, MAP_SHARED, shm_fd, (i - 1) *
getpagesize());
         if (vect == MAP_FAILED) {
```

```
            perror(NULL);
            shm_unlink("myshm");
            return errno;
        }
        int k = 0;
        vect[k++] = x;
        while (x != 1)
        {
            if (x % 2 == 0)
                x = x / 2;
            else
                x = 3 * x + 1;
            vect[k++] = x;
        }
        printf("Done Parent %d Me %d\n", getppid(), getpid());
        //demapare
        munmap(vect, n);
        //inchid procesul
        return 0;
    }
}
        for(int i=1;i<argc;i++)
            wait(NULL);
        for(int i=1;i<argc;i++)
        {
            int j;
            j=getpagesize()*(i-1)/sizeof(int);
            printf("%d: ", p[j] );
            while(p[j]>1)
            {
                printf("%d ", p[j]);
                j++;
            }
            printf(" 1\n");
        }
        printf("Done parent %d Me %d\n", getppid(), getpid());
        //demapare
        munmap(p, n);


return 0;
```

```
}
//lab6_1.c
void *invers(void *arg)
{
   //conversie explicita in char
   char *sir=(char*)arg;
   //trebuie sa si aloc spatiu pentru noul sir
   char * invers=(char*)malloc(strlen(sir));
   int j=0;
   for(int i=strlen(sir)-1; i>=0;i--)
   {
      invers[j]=sir[i];
      j++;
   }
   //fac inapoi conversie la void pointer
   return (void*)invers;
}
int main(int argc, char*argv[])
{
   char *sir=argc[1];
   p_thread thr;
   //NULL las sistemul de operare sa seteze atributele
   if(pthread_create(&thr, NULL, invers, sir))
   {
      perror(NULL);
      return errno;
   }
   void *sir_invers=NULL;
   if(pthread_join(thr, &sir_invers))
   {
      perror(NULL);
      return errno;
   }
   printf("%s\n", (char *)sir_invers);
   return 0;

}
//lab6_2.c
int a[5][5], b[5][5], c[5][5];
int m=5, n=5, p=5;
//structura pentru ca pot sa transmit
```

```
//o singura variabila ca parametru
typedef struct
{
   int x,y;
}s;
void *produs (void *arg)
{
   //convertesc pointerul void al argumentului
   //intr-unul de tipul s
   s *z=(s*)arg;
   int *r=(int*)malloc(sizeof(int));
   *r=0;
   for(int l=0;l<p;l++)
   {
      *r=*r+a[z->x][l] *b[l][z->y];
   }
   return (void *)r;
}
int main()
{
   int l=0, q=1;
   //matricea rezultata va avea m*n elemente
   //aloc spatiu pentru retinerea celor m*n thread-uri
   pthread_t *thr=(pthread_t*)malloc(sizeof(pthread_t)*m*n);
   //matricele a are nr de la 1 la m*p
   for(int i=0;i<m;i++)
      for(int j=0;j<n;j++)
      {
         a[i][j]=q;
         q++;
      }
   q=1;
   //matricea b are nr de la 1 la p*n
   for(int i=0;i<p;i++)
      for(int j=0;j<n;j++)
      {
         b[i][j]=q;
         q++;
      }
   //creez cele m*n threaduri
   for(int i=0;i<m;i++)
```

```
      for(int j=0;j<n;j++)
      {
         s*z=(s*)malloc(sizeof(s));
         z->x=i;
         z->y=j;
         if(pthread_create(&thr[l], NULL, produs, z))
         {
            perror(NULL);
            return errno;
         }
         l++;
      }
   //dau join celor m*n thread-uri
   l=0;
   for(int i=0;i<m;i++)
      for(int j=0;j<n;j++)
      {
        void * val=NULL;
         if(pthread_join(thr[l], &val))
         {
            perror(NULL);
            return errno;
         }
         //completare matrice rezultat
         c[i][j]=*((int*)val);
         l++;
      }
   //afisare rezultat
   for(int i=0;i<m;i++)
   {
      for(int j=0;j<n;j++)
         printf("%d ", c[i][j]);
      printf("\n");
   }
   return 0;

}
//lab7_1.c
#define MAX_RESOURCES 5
int av_res=MAX_RESOURCES;
//global ca sa aiba toate procesele acces la el
```

```
pthread_mutex_t mtx;
int decrease (int count)
{
   //lasa doar un singur proces sa treaca
   pthread_mutex_lock(&mtx);
   if(av_res <count)
   {
      pthread_mutex_unlock(&mtx);
      return -1;
   } else
   {
      av_res-=count;
      printf("Got %d resources %d remaining\n", count, av_res);

   }
   pthread_mutex_unlock(&mtx);
   return 0;
}
int increase(int count)
{
   pthread_mutex_lock(&mtx);
   av_res+=count;
   printf("Realeased %d resources %d remaining\n", count, av_res);
   pthread_mutex_unlock(&mtx);
}
void * resurse (void *v)
{
   int nr=*((int *)v);
   decrease(nr);
   increase(nr);
   return NULL;
}
int main()
{
   prthread_t thr[5];
   printf("MAX_RESOURCES=%d\n", MAX_RESOURCES);
   int i;
   if(pthread_mutex_init(&mtx, NULL))
   {
      perror(NULL);
      return errno;
```

```
    }
    //creez cele 5 thread-uri
    for(i=0;i<5;i++)
    {
        int *nr_res=(int*)malloc(sizeof(int));
        (*nr_res)=i+1;
        if(pthread_create(&thr[i], NULL, resurse, nr_res))
        {
            perror(NULL);
            return errno;
        }
    }
    //dau join celor 5
    for(i=0;i<5;i++)
    {
        if(pthread_join(thr[i], NULL))
        {
            perror(NULL);
            return errno;
        }
    }
    //distrugere mutex la final
    pthread_mutex_destroy(&mtx);
    return 0;
}
//lab7_2.c
int n=5,nr=0;
pthread_mutex_t mtx;
sem_t sem;
typedef struct
{
    pthread_t thr;
    int id;
}thread;
void barrier_point()
{
    pthread_mutex_lock(&mtx);
    nr++;
    pthread_mutex_unlock(&mtx);
    int i;
    int l=nr;
```

```
    if(l<n)
        sem_wait(&sem);//incrementeaza pana la n
    else
        for(i=1;i<=n-1;i++)
            sem_post(&sem);//decrementeaza pana la 1
}
void* tfun(void* v)
{
    //conversie
    int *tid= (int *)v;//id-ul procesului curent
    printf("%d reached the barrier\n", *tid);
    barrier_point();
    printf("%d passed the barrier\n", *tid);
    return NULL;
}
int main()
{
    thread t[5];//5 elemente de tipul structurii
    int i;
    //initializare mutex
    if(pthread_mutex_init(&mtx, NULL))
    {
        perror(NULL);
        return errno;
    }
    //initializare semafor
    //al doilea 0 spune daca vreau sa folosesc
    // semaforul in mai multe procese
    if(sem_init(&sem, 0, 0))
    {
        perror(NULL);
        return errno;
    }
    //creare thread-uri
    for(i=0;i<n;i++)
    {
        t[i].id=i+1;
        if(pthread_create(&t[i].thr, NULL, tfun, &t[i].id))
        {
            perror(NULL);
            return errno;
```

```
      }
   }
   //dau join thread-urilor
   for(i=0;i<n;i++)
      if(pthread_join(t[i].thr, NULL))
      {
         perror(NULL);
         return errno;
      }
   //distrug mutex-ul
   pthread_mutex_destroy(&mtx);
      //distrug semaforul
   sem_destroy(&sem);
   return 0;
}
//lab3_2.c
int sys_khello(struct proc *p, void *v, register_t *retval)
{
   struct sys_khello_args *uap=v;//citire argumente de intrare
   //nu e mallocul obisnuit din C
   char *kbuf=malloc(100, M_TEMP, M_WAITOK);
   //pune in el cat a reusit sa copieze
   size_t done;
   //macroul SCARG incarca din registrii
   //mutare in spatiul de adresare a kernelului a bufferului
   copyinstr(SCARG(uap, msg), kbuf, 100, &done);

   printf("Hello from kernel, %s! done=%ld\n", kbuf, done );

   //retval return catre user
   *retval=done;
   free(kbuf, M_TEMP, 100);
   return 0;//return catre kernel
}
//ex2.c
int main()
{
   char *nume="Oana";
   int ret;
   ret=syscall(331, nume);
   printf("Am primit din kernel %d\n", ret);
```

```
    return 0;
}
//lab3_3.c
int sys_kbuf(struct proc *p, void * v, register *retval)
{
    struct sys_khello_args *uap=v;//citire argumente de intrare

    size_t count=SCARG(uap, count);
    char * kbuf=malloc(count, M_TEMP, M_WAITOK);
    size_t bytesRead, bytesWritten;
    int state=copyinstr(SCARG(uap, source), kbuf, count, &bytesRead);
    if(state==EFAULT)
    {
        printf("eroare la copierea din sursa\n");
        free(kbuf, M_TEMP, count);
        return state;
    }
    state=copyoutstr(kbuf, SCARG(uap, target), bytesRead, &bytesWritten);
    target[byestWritten]='\0';
    if(state=EFAULT)
    {
        printf("eroare la trasnmiterea\n");
        free(kbuf, M_TEMP, count);
        return state;
    }
    *retval=bytesWritten;
    free(kbuf, M_TEMP, count);
    return 0;
}
//ex3.c
int main()
{
    char * source="ana are mere";
    char * dest=malloc(100);
    int rez=syscall(332, source, dest, 3);
    printf("%d octeti copiati: %d\n", res, dest);
    free(dest);
    return 0;
}
```