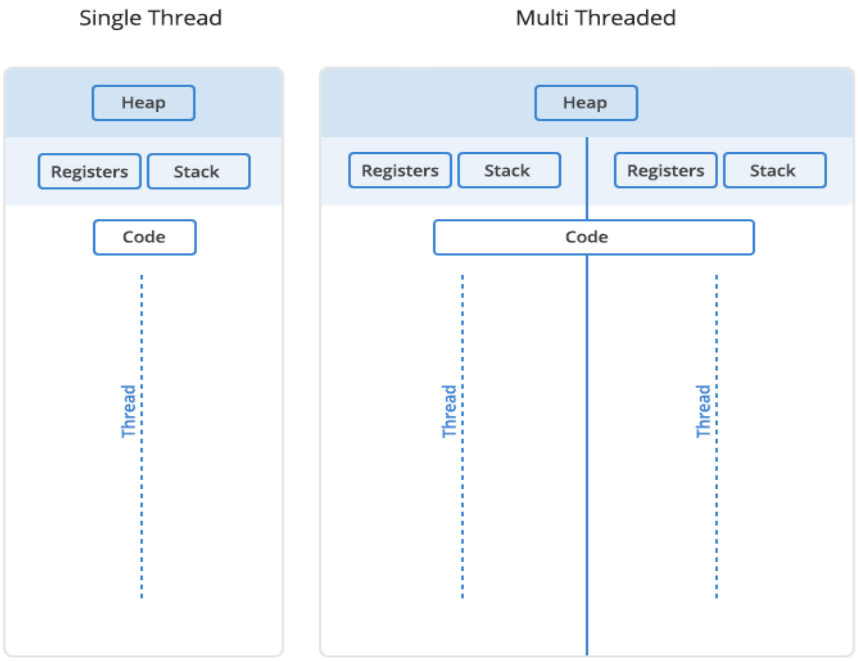
Seminary 5

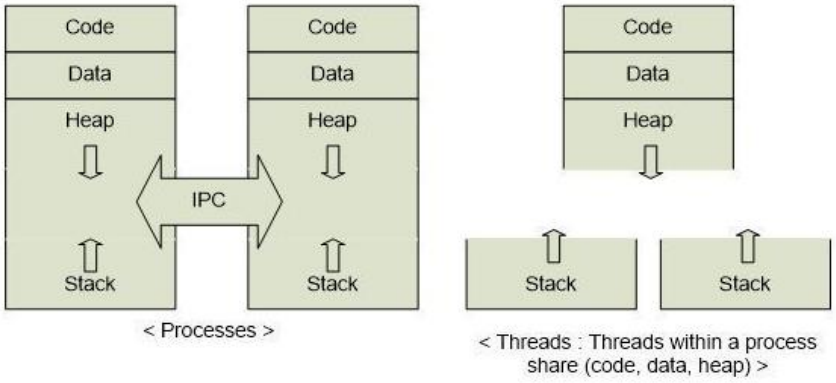
Thread, Mutex, RWLock, Conditional variable

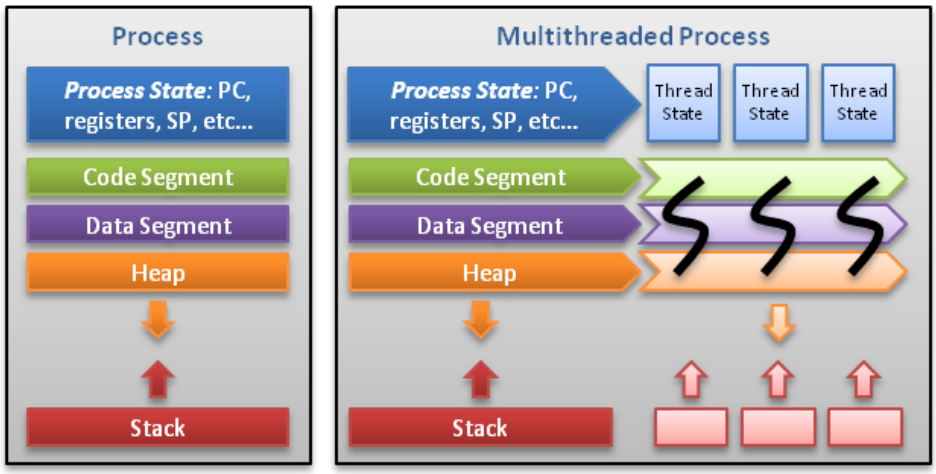
Header file	<pthread.h>
Compile using library	-pthread
Thread function definition	void * work(void* a)
Data types	pthread_t pthread_mutex_t pthread_cond_t pthread_rwlock_t sem_t
Thread Functions	int **pthread_create**(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg); int **pthread_join**(pthread_t thread, void **retval); void **pthread_exit**(void *retval);
Mutex Functions	int **pthread_mutex_init**(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr); //or pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; int **pthread_mutex_lock**(pthread_mutex_t *mutex); int **pthread_mutex_unlock**(pthread_mutex_t *mutex); int **pthread_mutex_destroy**(pthread_mutex_t *mutex);
Conditional variable functions	pthread_cond_init pthread_cond_wait pthread_cond_signal pthread_cond_broadcast pthread_cond_destroy
R/W Lock Functions	int **pthread_rwlock_init**(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr); //or pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER; int **pthread_rwlock_rdlock**(pthread_rwlock_t *rwlock); int **pthread_rwlock_wrlock**(pthread_rwlock_t *rwlock); int **pthread_rwlock_unlock**(pthread_rwlock_t *rwlock); int **pthread_rwlock_destroy**(pthread_rwlock_t *rwlock);
Semaphore Functions	sem_init sem_wait sem_post sem_destroy

Threads

☺ "Threads are just boneless processes." - Mahatma Gandhi







Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between allthreads.

© Alfred Park, http://randu.org/tutorials/threads

A single process can contain multiple threads, all of which are executing the same program. These threads share the same global memory (data and heap segments), but each thread has its own stack (automatic variables).

Threads share a range of other attributes: PID, PPID, open file descriptors, locks, signal dispositions, etc. Threads do NOT share (differ in): thread ID, errno variable, scheduling/priority, etc.

1. Write a program that receives strings as command line arguments and uses threads to capitalise each word. We'll create for each argument a separate thread that will capitalise the initial letter.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAXLINIE 1000
pthread_t tid[100]; //we need to refer to each thread to join them

void* ucap(void* numei) {
    printf("Thread start: %ld ...> %s\n", pthread_self(), (char*)numei);

    char numeo[100];
    strcpy(numeo, (char*)numei);
    if ( numeo[0]>=`a` && numeo[0]<=`z`)
        numeo[0]+='A'-'a';

    printf("Thread finished: %ld > %s\n", pthread_self(), (char*)numeo);
}
int main(int argc, char* argv[]) {
    int i;
    for (i=1; argv[i]; i++) {
        pthread_create(&tid[i], NULL, ucap, (void*)argv[i]);
        printf("Thread created: %ld ...> %s\n", tid[i], argv[i]);
    }
    for (i=1; argv[i]; i++) pthread_join(tid[i], NULL);
    printf("All threads finished\n");
}
```

Compile: `gcc -pthread capit.c`, and run `./a.out f1 f2 . . .`

If your system is missing manual pages for thread, mutex etc. function you can install it:

```
sudo apt-get install manpages-posix manpages-posix-dev

sudo apt-get install glibc-doc
```

The return value of a thread

2. Solve the problem of adding 4 numbers in parallel, but using two threads – one that calculates the sum of first two, another that calculated the sum of last two, and main program receives the calculated values and uses the partial sums to compute the total sum. For the return value we use `pthread_exit` and `pthread_join` (second argument) together with a pointer to a `return_val` which can be, as in our case, a struct.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAXLINIE 1000

pthread_t tid[100]; //we need to refer to each thread to join them
int a[]={1,2,3,4,5,6,7,8,9,10};

typedef struct { int ps; } response;

void* partial(void* id) {
        int nr= *(int *)id;

        response * r= malloc(sizeof(response));

        printf("Thread start: %ld ...> %d\n", pthread_self(), nr);

        r->ps=a[2*nr]+a[2*nr+1];

        printf("Thread finished: %ld > %d\n", pthread_self(), nr);

        pthread_exit(r);
}

int main(int argc, char* argv[]) {
        response  *r0, *r1;
        int tnr[100];
        int i=0;

        for (i=0; i<100; i++) {
                tnr[i]=i;
        }

        pthread_create(&tid[0], NULL, partial, (void*)&tnr[0]);
        pthread_create(&tid[1], NULL, partial, (void*)&tnr[1]);

        pthread_join(tid[0],  &r0);
        pthread_join(tid[1],  &r1);

        printf("All threads finished\n");

        printf("Total sum is: %d \n", ((response *)r0)->ps + r1->ps);

        free(r0);
        free(r1);

        return 0;
}
```

Homework – generalise the problem above to work for performing the parallel addition of 10 or more numbers n, using n/2 threads that compute the sum of pairs from index 2t and 2t+1, where t is the thread id. Use a separate for each section – to create the threads, to join the threads, to free memory.

Mutex

Why do we need such a thing? Synchronisation when using shared resources and we have **race conditions** and at least a **critical section**. The solution must ensure that:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

3. Write a program that has a global variable count and 1000 threads. Each thread increments count 1000 times. The following code will output different numbers in each execution.

```
#include <pthread.h>
#include <stdio.h>
int count = 0;
pthread_t tid[1000];

void* inc(void* nume) {
    for (int i = 0; i < 1000; i++) {
        int temp = count; temp++; count = temp;
    }
}

int main(int argc, char* argv[]) {
    int i;
    for (i=0; i < 1000; i++)
        pthread_create(&tid[i], NULL, inc, NULL);

    for (i=0; i < 1000; i++) pthread_join(tid[i], NULL);
    printf("count=%d\n", count);
}
```

What happens, why we have sometimes 1000000, sometimes 991000?

The three instructions inside the **for** of the **inc** thread functions are mixing up with those from the other threads running. What happens is that sometimes a thread reads a value that was already modified and changes it, causing loss of increments performed by other threads. This happens even if we replace

```
int temp = count; temp++; count = temp;
```

with `count++;`

because the count++ in itself is performed by the processor in 3 distinct steps:

(1) read/copy the value of count in registers,
(2) do the incremental in registers,
(3) Copy/move the new value from registers to the address of the variable count.

Any thread can be interrupted in its execution between the three operations, and meanwhile the value of n can change, while the thread will use the old read value when continuing its execution.

To ensure that whenever a thread reads a variable and need to modify it no other thread can read it until the first one finishes, we use a mutex.

We declare a global mutex: `pthread_mutex_t exclusive= PTHREAD_MUTEX_INITIALIZER;`

Alternativelu we may initialize it in main: `pthread_mutex_init(&exclusive, NULL);`

Surround the **critical code section** with lock/unlock:

```
pthread_mutex_lock(&exclusive);

int temp = count; temp++; count = temp;

pthread_mutex_unlock(&exclusive);
```

Deallocate at the end:

```
pthread_mutex_destroy(&exclusive);
```

4. Given n pairs of command line arguments which are integer numbers, computer how many pairs have (a) and even sum, (b) and odd sum, (c) at least one the arguments is 0 or nonnumerical. We will create a thread for each pair, and three global variables for counting the three conditions, which will be accessed exclusively by each thread. We should be using a separate mutex for each – we want to correctly synchronise and have as much as possible performed in parallel.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define MAXLINIE 1000

typedef struct {char*n1; char*n2;} PERECHE;

pthread_t tid[100];
PERECHE pair[100];

pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mtxeven = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mtxodd = PTHREAD_MUTEX_INITIALIZER;

int pare = 0, impare = 0, nenum = 0;


void* computepairs(void* pair) {

    int n1 = atoi(((PERECHE*)pair)->n1);
    int n2 = atoi(((PERECHE*)pair)->n2);

    if (n1 == 0 || n2 == 0) {
        pthread_mutex_lock(&mut);
        nenum++;
        pthread_mutex_unlock(&mut);
    }
    else if ((n1 + n2) % 2 == 0) {
        pthread_mutex_lock(&mtxeven);
        pare++;
        pthread_mutex_unlock(&mtxeven);
    }
    else {
        pthread_mutex_lock(&mtxodd);
        impare++;
        pthread_mutex_unlock(&mtxodd);
    }
}
```

```
int main(int argc, char* argv[]) {
    int i, p, n = (argc-1)/2;

    for (i = 1, p = 0; p < n; i += 2, p++) {
        pair[p].n1 = argv[i];
        pair[p].n2 = argv[i+1];
        pthread_create(&tid[p], NULL, computepairs, (void*)&pair[p]);
        // We need to allocate separate memory for each thread argument!!
// Trying to reuse this variable will mess-up the code and the threads will
//not receive their arguments correctly. Incorect to say ..., (void*)&pair);
    }
    for (i=0; i < n; i++)
        pthread_join(tid[i], NULL);

    printf("pairs=%d even=%d odd=%d nonnumeric=%d\n",n,pare,impare,nenum);

    pthread_mutex_destroy(&mut);

    pthread_mutex_destroy(&mtxodd);

    pthread_mutex_destroy(&mtxeven);

    return 0;
}
```

a. What happens if we reuse variable pair instead of an array of pairs, one for each thread?
b. What happens if we place create and join in the same for loop?

```
for (i = 1, p = 0; p < n; i += 2, p++) {
        pair[p].n1 = argv[i];
        pair[p].n2 = argv[i+1];
        pthread_create(&tid[p], 0, computepairs, (void*)&pair[p]);
        pthread_join(tid[i], NULL);

}
```

c. Why we use 3 mutexes instead of 1?

Read Write Lock – Multiple readers, few writers problem

A read lock `pthread_rwlock_rdlock` allows any number or readers to access the resources for reading only, but no writers are allowed. A write lock `pthread_rwlock_wrlock` allows only one writer to access the resource, no other writers/readers allowed.

It is the classical database access problem – a lot of people read, very few modify. For example online catalogue of grades/attendance – edited by the professor in rare occasions, read often by many students.

5. Write a program to simulate this problem of multiple readers reading symoultaneously and only one writer at a time.

States of a writer thread (S):	States of a reader thread (C):
-3 writer not started, -2 writer managed to write and will sleep, -1 waits for readers to finish their operations, -0 writer is writing.	-3 reader not started yet, -2 reader managed to read and will sleep, -1 readers waits for writers to finish, 0 reader is reading.

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#define C 7 //readers
#define S 2 //writers
#define CSLEEP 2
#define SSLEEP 3

pthread_t tid[C + S]; //some writer threads, some reading threads
int c[C], s[S], nt[C + S];
pthread_rwlock_t rwlock;
pthread_mutex_t exclusafis;

//print states of readers and writers
void afiseaza() {
    int i;
    pthread_mutex_lock(&exclusafis);
    for (i = 0; i < C; i++) printf("C%d_%d\t",i, c[i]);
    for (i = 0; i < S; i++) printf("S%d_%d\t",i, s[i]);
    printf("\n");
    fflush(stdout);
    pthread_mutex_unlock(&exclusafis);
}

//reader thread function
void* cititor(void* nrc) {
    int indc = *(int*)nrc;
    for ( ; ; ) {
        c[indc] = -1; // Waits to read if locked by a writer

        pthread_rwlock_rdlock(&rwlock);
        c[indc] = 0; // Reads
        afiseaza();
        sleep(1 + rand() % CSLEEP); //simulate reading operation time
        c[indc] = -2; // Reading finished and sleeps
        pthread_rwlock_unlock(&rwlock);

        sleep(1 + rand() % CSLEEP);
    }
}

//writer thread function
void* scriitor (void* nrs) {
    int inds = *(int*)nrs;
    for ( ; ; ) {
        s[inds] = -1; // Waiting to write

        pthread_rwlock_wrlock(&rwlock);
        s[inds] = 0; // Writes
        afiseaza();
        sleep(1 + rand() % SSLEEP); //similate writing operation time
        s[inds] = -2; // Write operation finished and sleeps
        pthread_rwlock_unlock(&rwlock);

        sleep(1 + rand() % SSLEEP);
    }
}
```

```
int main() {

    pthread_rwlock_init(&rwlock, NULL);
    pthread_mutex_init(&exclusafis, NULL);

    int i;
    for (i = 0; i < C; c[i] = -3, nt[i] = i, i++); // -3 : State of Not started
    for (i = 0; i < S; s[i] = -3, nt[i + C] = i, i++);

    //launch threads
    for (i = 0; i < C; i++) pthread_create(&tid[i], NULL, cititor, &nt[i]);
    for (i = C; i < C + S; i++) pthread_create(&tid[i], NULL, scriitor, &nt[i]);

    for (i = 0; i < C + S; i++) pthread_join(tid[i], NULL);

    pthread_rwlock_destroy(&rwlock);
    pthread_mutex_destroy(&exclusafis);
}
```

6. 100 participants in the BEST COMPUTER SCIENCE MEMES contest are waiting for the results. Three sponsors from big companies provide each a prize. Model this problem using threads, in which each competitor checks randomly every few seconds for the announced winners, until they hear their number or all three winners were announced. The jury sponsors take a larger number of seconds to deliberate and state their winners.

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#define C 20 //competitors
#define S 3 //sponsors
#define CSLEEP 2
#define SSLEEP 3

pthread_t tid[C + S]; //some writer threads, some reading threads
int nt[C + S]; // if we wnt to pass ids instead of i

int nrp=0;
int p[S];


pthread_rwlock_t rwlock;


//writer thread function
void* sponsor (void* i) {
    int sponsor = (int)i;
    // or *(int *) nt if we use the array version nt[] to pass arg thread id

    sleep(3 + rand() % SSLEEP); //deliberate

    pthread_rwlock_wrlock(&rwlock);
    p[nrp]=rand()%(C+1);
    nrp++;
    printf("Winner is: %d \n", p[nrp-1]);
    pthread_rwlock_unlock(&rwlock);

}
```

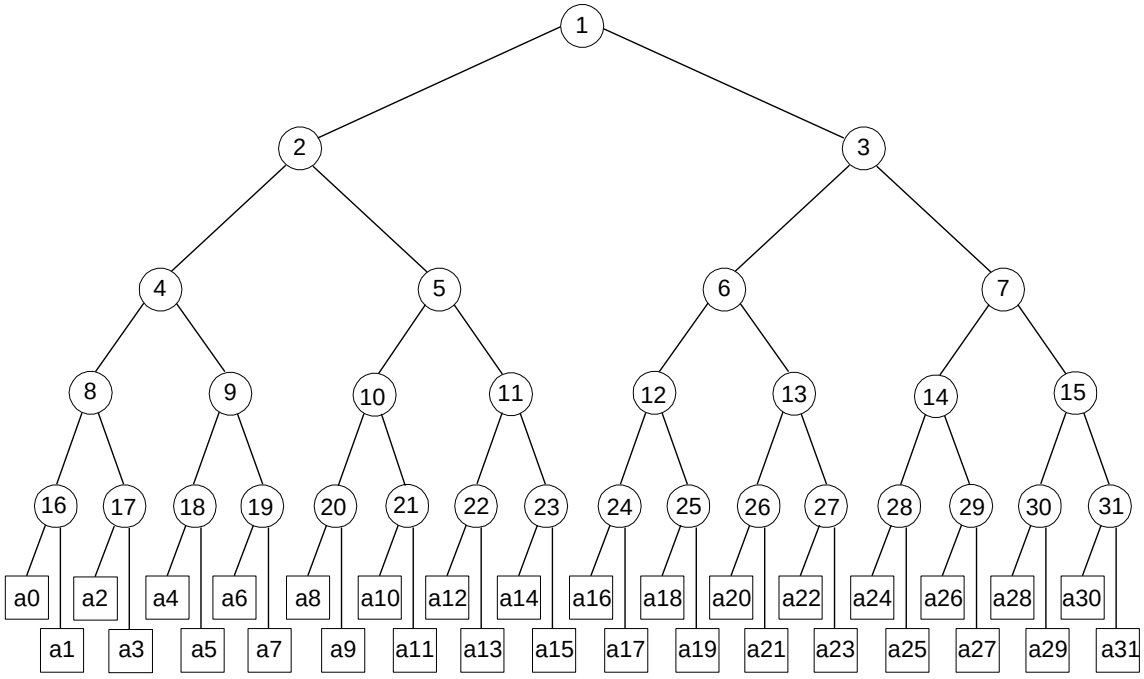
```
        //reader thread function
        void* competitor(void* nrc) {
            int ct = (int)nrc;
            int f=0, i=0;

            while (f==0 && nrp<S){

                      pthread_rwlock_rdlock(&rwlock);
                      printf("%d is cheking the winners \n", ct);

                      for ( i=0; i<nrp; i++){
                                  if (p[i]==ct) {
                                     f=1; printf("Winner me %d!!! \n", ct);
                                  }
                      }
                      pthread_rwlock_unlock(&rwlock);
                      sleep(1 + rand() % CSLEEP);

           }


           //do a last check for the last winner announced
           pthread_rwlock_rdlock(&rwlock)
           for ( i=0; i<nrp; i++){
                       if (p[i]==ct) {
                               f=1; printf("Winner me %d!!! \n", ct);
                       }
           }
           pthread_rwlock_unlock(&rwlock);
        }



        int main() {

            pthread_rwlock_init(&rwlock, NULL);
            pthread_mutex_init(&exclusafis, NULL);

            int i;
            // for (i = 0; i < S + C; nt[i] = i, i++);
            //launch threads – i warding integer to pointer cast; use instead &nt[i]

            for (i = 0; i < C; i++) pthread_create(&tid[i], NULL, competitor, i);
            for (i = C; i < C + S; i++) pthread_create(&tid[i], NULL, sponsor, i);

            for (i = 0; i < C + S; i++) pthread_join(tid[i], NULL);

            pthread_rwlock_destroy(&rwlock);
            pthread_mutex_destroy(&exclusafis);

            return 0;
}
```

7. Similar problem could be the case of Worldometer statistics website, people loading the page to see the number of COVID-19 cases (readers) and authorities updating with new cases (writers) from time to time. Similar to example 5, readers and writers are operating in a continuous loop with random sleeps between reads and writes. Model and implement this problem using threads and RWlocks to maximise efficiency.
   a. Try the implementation without using external sources (only terminal and manual).
   b. Experiment with different number of threads for read/write and various intervals for the random sleeps. Too many readers may not leave resources for writers too often… see what happens.
   c. Replace the RWlock with Mutex and observe the difference over a period of time.

Hardcore training with threads

Add in parallen n numbers using threads, given a global array *a[0], a[1], . . . a[n-1].*

Idea: create a structure tree hierarchy of threads that wait for other threads to compute partial sums. For example for n=32 we need 31 threads. For example thread 16 calculates the sum of a[0] and a[1], and so on, thread 31 computeas the sum of a[30]+a[31]. The next layer of threads use these partial sums, for example thread 8 computes the partial sum from threads 16 and 17, so the sum of the first 4 numbers.




```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int n, m; // n = numarul de operanzi; m = min {2^k >= n}
int* a; // valoarea 1 pentru pana la n-1,  0 de la n la m-1
pthread_t *tid; // id-urile threadurilor; -1 thread nepornit
pthread_mutex_t print = PTHREAD_MUTEX_INITIALIZER; // Printare exclusiva

// Rutina thread-ului nr i de adunare
void* aduna(void* pi) {
    int i, j, sa, da, st = 0, dr = 0, k;
    i = *(int*)pi; // Retine numarul threadului
    if (i < m / 2) {
        st = 2 * i; // Retine fiul stang
        dr = st + 1; // Retine fiul drept
        while (tid[st] == -1);     // Asteapta sa inceapa fiul stang
        // while (tid[st] == -1) sleep(1); // poate asa!
        // Cel mai sanatos este să se utilizeze un set de variabile
coditionale
        // care sa semnaleze pornirile threadurilor.
        while (tid[dr] == -1);     // Asteapta sa inceapa fiul drept
        // while (tid[dr] == -1) sleep(1); // poate asa!
        pthread_join(tid[st], NULL); // Asteapta sa se termine fiul stang
```

```
            pthread_join(tid[dr], NULL); // Asteapta sa se termine fiul drept
        }
        for (j = m; j > i; j /= 2); // Determina fratele cel mic
        for (k = j, sa = 0; k < i; k++) sa += m / j; // operand stang
        da = sa + m / j / 2; // operand drept
        a[sa] += a[da]; // Face adunarea proppriu-zisa
        pthread_mutex_lock(&print);// Asigura printare exclusiva
        printf("Thread %d: a[%d] += a[%d]", i, sa, da);
        if (st > 0) printf(" (dupa fii %d %d)\n", st, dr); else printf("\n");
        pthread_mutex_unlock(&print);
    }

    // Functia main, in care se creeaza si lanseaza thread-urile
    int main(int argc, char* argv[]) {
        n = atoi(argv[1]); // Numarul de numere de adunat
        for (m = 1; n > m; m *= 2); // m = min {2^k >= n}
        int* pi;
        int i;
        a = (int*) malloc(m*sizeof(int)); // Spatiu pentru intregii de adunat
        pi = (int*) malloc(m*sizeof(int)); // Spatiu pentru indicii
    threadurilor
        tid = (pthread_t*) malloc(m*sizeof(pthread_t)); // id-threads
        for (i = 0; i < n; i++) a[i] = 1; // Aduna numarul 1 de n ori
        for (i = n; i < m; i++) a[i] = 0; // Completeaza cu 0 pana la m
        for (i = 1; i < m; i++) tid[i] =-1; // Threadurile sunt inca nepornite
        for (i = 1; i < m; i++) pi[i] = i; // Threadurile sunt inca nepornite
        for (i = 1; i < m; i++)
            // De ce folosim mai jos &pi[i] in loc de &i? vezi un exemplu
    precedent!
            pthread_create(&tid[i], NULL, aduna, (void*)(&pi[i])); //
    Threadul i
        pthread_join(tid[1], NULL); // Asteapta dupa primul thread
        printf("Terminat adunarile pentru n = %d. Total: %d\n", n, a[0]);
        free(a); // Eliberaza tabloul de numere
        free(pi); // Elibereaza tabloul de indici de threaduri
        free(tid); // Elibereaza tabloul de id-uri de threaduri
    }
```