# Pthreads; sincronizări cu mutex, cond, barrier

## Principalele tipuri de date şi funcţii de lucru cu threaduri

Tabelul următor prezintă principalele fişiere header, tipuri de date şi funcţii care lucrează cu threaduri:

|  |  |
| --- | --- |
| **Fişere header** | <pthread.h> |
| **Specificare biblioteci** | -pthread |
| **Tipuri de date** | pthread\_t  pthread\_mutex\_t  pthread\_cond\_t  pthread\_barrier\_t  pthread\_rwlock\_t  sem\_t |
| **Funcţii de creare thread**  **şi aşteptare terminare** | pthread\_create  pthread\_join  pthread\_exit |
| **Variabile mutex** | pthread\_mutex\_init  pthread\_mutex\_lock  pthread\_mutex\_unlock  pthread\_mutex\_destroy |
| **Variabile condiţionale** | pthread\_cond\_init  pthread\_cond\_wait  pthread\_cond\_signal  pthread\_cond\_broadcast  pthread\_cond\_destroy |
| **Bariere** | pthread\_barrier\_init  pthread\_barrier\_destroy  pthread\_barrier\_wait |
| **Variabile reader/writer** | pthread\_rwlock\_init  pthread\_rwlock\_wrlock  pthread\_rwlock\_rdlock  pthread\_rwlock\_unlock  pthread\_rwlock\_destroy |
| **Semafoare** | sem\_init  sem\_wait  sem\_post  sem\_destroy |
| **Funcţia de descriere a acţiunii threadului** | void \* work(void\* a) |

## Capitalizarea cuvintelor dintr-o listă de fişiere text

Dorim sa transformam un fişier text într-un alt fişier text, cu acelaşi conţinut, dar în care toate cuvintele din el sa înceapă cu literă mare. Un astfel de program / procedura o vom numi capitalizare si se apeleaza furnizand doi parametri: fisierintrare fisieriesire

Ne propunem să prelucrăm simultan mai multe astfel de fisiere. Programul primeste numele a **n** fisiere de intrare, iar fisierele de iesire vor primi acelasi nume la care i se adauga terminatia .CAPIT. Programul va crea câte un thread pentru fiecare pereche de fişiere. Sursa capitalizari.c este:

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAXLINIE 1000

pthread\_t tid[100];

// pthread\_t tid; // Vezi comentariul de la sfârşitul sursei

void\* ucap(void\* numei) {

printf("Threadul: %ld ...> %s\n", pthread\_self(), (char\*)numei);

FILE \*fi, \*fo;

char linie[MAXLINIE], numeo[100], \*p;

strcpy(numeo, (char\*)numei);

strcat(numeo, ".CAPIT");

fi = fopen((char\*)numei, "r");

fo = fopen(numeo, "w");

for ( ; ; ) {

p = fgets(linie, MAXLINIE, fi);

linie[MAXLINIE-1] = '\0';

if (p == NULL) break;

if (strlen(linie) == 0) continue;

linie[0] = toupper(linie[0]); // Cuvant incepe in coloana 0

for (p = linie; ; ) {

p = strstr(p, " ");

if (p == NULL) break;

p++;

if (\*p == '\n') break;

\*p = toupper(\*p);

}

fprintf(fo, "%s", linie);

}

fclose(fo);

fclose(fi);

printf("Terminat threadul: %ld ...> %s\n", pthread\_self(), (char\*)numei);

}

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

int i;

for (i=1; argv[i]; i++) {

pthread\_create(&tid[i], NULL, ucap, (void\*)argv[i]);

// pthread\_create(&tid, NULL, ucap, (void\*)argv[i]); // Vezi comentariul de la sfârşitul sursei

printf("Creat threadul: %ld ...> %s\n", tid[i], argv[i]);

}

for (i=1; argv[i]; i++) pthread\_join(tid[i], NULL);

// for (i=1; argv[i]; i++) pthread\_join(tid, NULL); // Vezi comentariul de la sfârşitul sursei

printf("Terminat toate threadurile\n");

}

Compilarea se face: gcc -pthread capitalizari.c, iar rularea ./a.out f1 f2 . . .

In sursa de mai sus sunt trei linii comentariu, în care, în loc de tid[i] se foloseşte tid. Este vorba de a folosi o singură variabilă în loc de a folosi un tablou în care să se folosească câte o intrare pentru fiecare thread în parte. Din punct de vedere sintactic, folosirea unei singure variabile este perfect valabilă. Insă, din punct de vedere al funcţionării, dacă se foloseşte o singură variabilă, atunci funcţiile pthread\_join vor aştepta numai după ultimul thread!

## Capitalizarea cuvintelor dintr-o listă de fişiere text - soluţia go

//Se capitalizeaza multithreading continutul unor fisiere:./capitalizeWord f1 f2 ...

package main

import (

"bufio"

"fmt"

"io"

"os"

"strings"

"bytes"

)

func ucap(numei string, finish chan<- struct{}) {

print(" Capitalizare: "+numei+"\n")

f,\_ := os.Open(numei)

fi := bufio.NewReader(f)

fo,\_ := os.Create(numei+".CAPIT")

for {

linie,err := fi.ReadString('\n')

if err == io.EOF { break }

r := 0

b := byte(linie[0])

u := string(bytes.ToUpper([]byte{b}))

linie = u+linie[1:]

for {

i := strings.IndexByte(linie[r:], ' ')

if i == -1 { break }

i = r + i + 1

r = i + 1

if r >= len(linie) { break }

b = byte(linie[i])

u = string(bytes.ToUpper([]byte{b}))

linie = linie[:i]+u+linie[(i+1):]

}

fmt.Fprint(fo, linie)

}

f.Close()

fo.Close()

print(" Terminat capitalizare "+numei+"\n")

finish <- struct{}{}

}

func main() {

finish := make(chan struct{})

for i :=1; i < len(os.Args); i++ { go ucap(os.Args[i], finish) }

for i :=1; i < len(os.Args); i++ { <- finish }

print("Terminat toate capitalizarile\n");

}

## Capitalizarea cuvintelor dintr-o listă de fişiere text - soluţia python

# Se capitalizeaza multithreading continutul unor fisiere:./capitalizeWord f1 f2 ...

import sys

import os

import threading

def ucap(numei):

print(" Capitalizare: "+numei+"\n")

fi = open(numei, "r")

fo = open(numei+".CAPIT", "w")

while True:

linie = fi.readline()

if linie == "": break

r = 0

linie = linie[0].upper()+linie[1:]

while True:

i = linie[r:].find(" ")

if i == -1: break

i = r + i + 1

r = i + 1

if r >= len(linie): break

linie = linie[:i]+linie[i].upper()+linie[(i+1):]

fo.write(linie)

fi.close()

fo.close()

print(" Terminat capitalizare "+numei+"\n")

def main():

t = []

for i in range(1, len(sys.argv)):

t.append(threading.Thread(target=ucap, args=(sys.argv[i],)))

t[i-1].start()

for i in range(1, len(sys.argv)):

t[i-1].join()

print("Terminat toate capitalizarile\n")

main()

## De ce sunt necesare variabilele mutex?

Să construim un program care are o variabilă globală **count** şi 1000 de threaduri. Fiecare thread incrementează de câte 1000 de ori variabila count. In acest scop să considerăm programul următor:

#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);

}

Evident, funcţia inc a threadului se poate scrie: **count += 1000**. In mod intenţionat am "prelungit" execuţia în funcţia threadului. Rezultatele unor execuţii repetate ale programului de mai sus sunt:

florin@ubuntu:~/pthreads$ gcc -pthread inc.c

florin@ubuntu:~/pthreads$ ./a.out

count=997000

florin@ubuntu:~/pthreads$ ./a.out

count=997000

florin@ubuntu:~/pthreads$ ./a.out

count=980000

florin@ubuntu:~/pthreads$ ./a.out

count=1000000

florin@ubuntu:~/pthreads$ ./a.out

count=991000

florin@ubuntu:~/pthreads$ ./a.out

count=996000

florin@ubuntu:~/pthreads$ ./a.out

count=980000

florin@ubuntu:~/pthreads$ ./a.out

count=1000000

florin@ubuntu:~/pthreads$ ./a.out

count=997000

florin@ubuntu:~/pthreads$ ./a.out

count=995000

florin@ubuntu:~/pthreads$

De ce acest comportament ciudat? Pentru că execuţia threadurilor se face în paralel, deci este posibil ca cele trei instrucţiuni ale corpului for din funcţia **inc** să se înterpătrundă, aşa încât două threaduri să memoreze succesiv valoarea lui **count** independent de reţinerea ei în variabila locală temp. Astfel, se pierde efectul unuia dintre threaduri.

Pentru a asigura derularea corectă, se impune utilizarea unui mutex:

Se declară variabila globală

pthread\_mutex\_lock(&exclusiv);

Corpul ciclului se va înlocui cu:

pthread\_mutex\_lock(&exclusiv);

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

pthread\_mutex\_unlock(&exclusiv);

## Câte perechi de argumente au suma număr par?

La linia de comandă se dau **n** perechi de argumente despre care se presupune ca sunt numere întregi si pozitive. Se cere numărul de perechi care au suma un număr par, numărul de perechi ce au suma număr impar si numărul de perechi în care cel putin unul dintre argumente nu este număr strict pozitiv.

Rezolvarea: Se va crea câte un thread pentru fiecare pereche. Trei variabile globale, cu rol de contoare, vor număra fiecare categorie de pereche.

**Important:** deoarece threadurile pot incrementa simultan unul dintre contoare, este necesar să se asigure accesul exclusiv la aceste contoare. De aceea, vom folosi o variabilă mutex care va proteja acest acces.

Intrebare: este corect sau nu să fie protejat fiecare contor printr-o variabilă mutex proprie? In exemplul nostru le protejăm simultan pe toate trei cu acelaşi mutex.

Sursa programului este:

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define MAXLINIE 1000

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

pthread\_t tid[100];

PERECHE pereche[100];

// PERECHE pereche; // Vezi comentariul de la sfârşitul sursei

pthread\_mutex\_t mut = PTHREAD\_MUTEX\_INITIALIZER;

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

void\* tippereche(void\* pereche) {

int n1 = atoi(((PERECHE\*)pereche)->n1);

int n2 = atoi(((PERECHE\*)pereche)->n2);

if (n1 == 0 || n2 == 0) {

pthread\_mutex\_lock(&mut);

nenum++;

pthread\_mutex\_unlock(&mut);

} else if ((n1 + n2) % 2 == 0) {

pthread\_mutex\_lock(&mut);

pare++;

pthread\_mutex\_unlock(&mut);

} else {

pthread\_mutex\_lock(&mut);

impare++;

pthread\_mutex\_unlock(&mut);

}

}

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

int i, p, n = (argc-1)/2;

for (i = 1, p = 0; p < n; i += 2, p++) {

pereche[p].n1 = argv[i];

pereche[p].n2 = argv[i+1];

pthread\_create(&tid[p], NULL, tippereche, (void\*)&pereche[p]);

// !! Apelul urmator nu este corect, Vezi explicatia la sfârşitul sursei

// pthread\_create(&tid[p], NULL, tippereche, (void\*)&pereche);

}

for (i=0; i < n; i++) pthread\_join(tid[i], NULL);

printf("perechi=%d pare=%d impare=%d nenum=%d\n",n,pare,impare,nenum);

}

In sursa de mai sus există un comentariu unde în loc de pereche[p] se foloseşte pereche. Este vorba de a folosi o aceaşi variabilă simplă în loc de a folosi un tablou în care să se folosească câte o intrare pentru fiecare thread în parte. Din punct de vedere sintactic, folosirea unei singure variabile este perfect valabilă. Insă, din punct de vedere al funcţionării, dacă se foloseşte o singură variabilă, se comite una dintre cele mai dese erori de logică în lucrul cu threaduri: Dacă se foloseşte o singură variabilă care transmite parametrul de intrare pentru toate apelurile ucap ale tuturor threadurilor, este posibil ca (PERECHE\*)pereche)->n1 sau (PERECHE\*)pereche)->n2 să nu preia intrările pregătite, ci să preia valori pregătite pentru threadul următor!

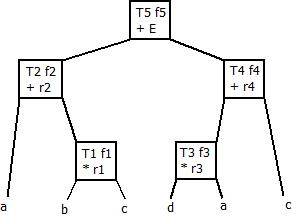
## Evaluarea expresie aritmetică operator / thread şi paralelizare maximă

Se dă expresia aritmetică

E = ( a + b \* c ) + ( d \* a + c )

unde a, b c, d sunt numere întregi. Se cere evaluarea acestei expresii executând fiecare operaţie într-un thread separat şi cu lansarea a câtor mai multe threaduri în acelaşi timp.

Pentru rezolvare, vom folosi cinci threaduri T1, T2, T3, T4, T5, variabilele intermediare r1, r3, r3, r4, E şi funcţiile de thread f1, f2, f3, f4, f5. Arborele din figura următoare ilustrează evaluarea acestei expresii:



In cele cinci pătrate sunt ilustrate cele cinci threaduri care contribuie la rezolvarea problemei. Pentru fiecare thread se indică numele threadului, funcţia care îi determină funcţionarea, operaţia executată şi variabila unde se depune rezultatul operaţiei.

Sursa programului este dată mai jos:

# include <stdio.h>

# include <pthread.h>

pthread\_t t[6];

int a=1, b=2, c=3, d=4, r1, r2, r3, r4, E;

void \* f1 (void \* x)

{r1 = b \* c;}

void \* f2 (void \* x)

{r2 = a + r1;}

void \* f3 (void \* x)

{r3 = d \* a;}

void \* f4 (void \* x)

{r4 = r3 + c;}

void \* f5 (void \* x)

{E = r2 + r4;}

int main() {

/\* O prima varianta de creare / join

pthread\_create(&t[1], NULL, f1, NULL);

pthread\_create(&t[2], NULL, f2, NULL);

pthread\_create(&t[3], NULL, f3, NULL);

pthread\_create(&t[4], NULL, f4, NULL);

pthread\_create(&t[5], NULL, f5, NULL);

pthread\_join(t[1], NULL);

pthread\_join(t[2], NULL);

pthread\_join(t[3], NULL);

pthread\_join(t[4], NULL);

pthread\_join(t[5], NULL);

Sfarsit varianta 1 - rezultat E=0! \*/

// Varianta corecta:

// T2 lansat dupa ce se termina T1 si T4 lansat dupa ce se termina T3

// T5 lansat numai dupa ce se termina T2 si T4

pthread\_create(&t[1], NULL, f1, NULL);

pthread\_create(&t[3], NULL, f3, NULL);

pthread\_join(t[1], NULL);

pthread\_join(t[3], NULL);

pthread\_create(&t[2], NULL, f2, NULL);

pthread\_create(&t[4], NULL, f4, NULL);

pthread\_join(t[2], NULL);

pthread\_join(t[4], NULL);

pthread\_create(&t[5], NULL, f5, NULL);

pthread\_join(t[5], NULL);

printf("E = %d\n", E);

}

Dacă se rulează prima variantă, în care nici un thread nu aşteaptă după altul, rezultatul este 0! Asta pentru faptul că threadurile care încarcă variabilele r1 - r4 nu apucă să o facă şi threadul T5 preia din ele valorile 0!

Varianta corectă este aceea în care T2 şi T4 aşteaptă să se termine mai întâi T1 şi T2, apoi T5 aşteaptă să se termine mai întâi T2 şi T4.

## De ce sunt necesare variabilele condiţionale?

In exemplul de mai sus unele threaduri au fost nevoite să aştepte terminarea altora pentru a îşi termina activitatea lor. Ce este de făcut dacă NU poate trece de un punct (să zicem **A**) al execuţiei până când un alt thread nu ajunge cu execuţia într-un punct (să zicem **B**). (In literatură fenomenul se numeşte **rendezvouz**). Să schematizăm acest fenomen prin următorul program cu două threaduri, în care evenimentul de rendezvouz este variabila considerăm un exemplu:

#include <pthread.h>

#include <stdio.h>

int eveniment = 0;

// - - -

pthread\_t tid[2];

void\* produceeveniment(void\* nume) {

// - - - Calcule - - -

// B

printf("B va produce evenimentul\n");

eveniment = 1;

printf("B a produs evenimentul\n");

// - - - alte calcule - - -

}

void\* asteaptaeveniment(void\* nume) {

// - - - Calcule - - -

// A

printf("A asteapta evenimentul\n");

while (eveniment == 0) {

;

}

printf("A a primit evenimentul\n");

// - - - alte calcule - - -

}

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

pthread\_create(&tid[1], NULL, asteaptaeveniment, NULL);

pthread\_create(&tid[0], NULL, produceeveniment, NULL);

pthread\_join(tid[0], NULL);

pthread\_join(tid[1], NULL);

}

In funcţie de ordinea de creare a celor două threaduri, efectul execuţiei va fi:

|  |  |
| --- | --- |
| A asteapta evenimentul  B va produce evenimentul  B a produs evenimentul  A a primit evenimentul | B va produce evenimentul  B a produs evenimentul  A asteapta evenimentul  A a primit evenimentul |

**Marea problemă** este aşteptarea evenimentului: **while (eveniment == 0){ ; };**

In modul de mai sus avem de-a face cu o **aşteptare activă**, threadul care aşteaptă ocupă procesorul cu execuţia lui **while**! Este de dorit ca până la apariţia evenimentului, threadul care aşteaptă **să nu ocupe procesorul**! (Din păcate, soluţia cu semnale, pe care am descris-o la procese NU MERGE, deoarece threadurile producător şi de aşteptare sunt în acelaşi proces.).

**Rezolvarea se face printr-o variabilă condiţională**. Pentru aceasta se declară o variabilă condiţională şi o variabilă mutex asociată ei:

pthread\_cond\_t variabila = PTHREAD\_COND\_INITIALIZER;

pthread\_mutex\_t mutexasociat = PTHREAD\_MUTEX\_INITIALIZER;

Secvenţa de provocare a evenimentului se înlocuieşte cu una din variantele (a doua variantă sincronizează şi tipărirea celor două mesaje):

|  |  |
| --- | --- |
| pthread\_mutex\_lock(&mutexasociat);  eveniment = 1;  pthread\_cond\_signal(&variabila);  pthread\_mutex\_unlock(&mutexasociat); | pthread\_mutex\_lock(&mutexasociat);  printf("B va produce evenimentul\n");  eveniment = 1;  pthread\_cond\_signal(&variabila);  printf("B a produs evenimentul\n");  pthread\_mutex\_unlock(&mutexasociat); |

Secvenţa de cedare a procesorului pe durata aşteptării evenimentului se înlocuieşte cu una din variantele (a doua variantă sincronizează şi tipărirea celor două mesaje):

|  |  |
| --- | --- |
| pthread\_mutex\_lock(&mutexasociat);  while (eveniment == 0) {  pthread\_cond\_wait(&variabila,  &mutexasociat);  }  pthread\_cond\_signal(&variabila);  pthread\_mutex\_unlock(&mutexasociat); | pthread\_mutex\_lock(&mutexasociat);  printf("A asteapta evenimentul\n");  while (eveniment == 0) {  pthread\_cond\_wait(&variabila,  &mutexasociat);  }  pthread\_cond\_signal(&variabila);  printf("A a primit evenimentul\n");  pthread\_mutex\_unlock(&mutexasociat); |

In funcţie de ordinea de creare a celor două threaduri, efectul execuţiei va fi:

|  |  |
| --- | --- |
| B va produce evenimentul  B a produs evenimentul  A asteapta evenimentul  A a primit evenimentul | A asteapta evenimentul  B va produce evenimentul  B a produs evenimentul  A a primit evenimentul |

## Bariera - exemplu

Pthread\_barrier este o construcție care permite ca mai multe threaduri independente să „aștepte” în spatele unei „bariere” pana când toate cele care aşteaptă ajung la barieră. Bariera este opțională în standardul POSIX, aşa că unele sisteme de operare nu o includ. Iată un exemplu simplu de utilizare a unei bariere:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include <time.h>

#define THREAD\_COUNT 4

pthread\_barrier\_t mybarrier;

void\* threadFn(void \*id\_ptr) {

int thread\_id = \*(int\*)id\_ptr;

int wait\_sec = 1 + rand() % 5;

printf("thread %d: Wait for %d seconds.\n", thread\_id, wait\_sec);

sleep(wait\_sec);

printf("thread %d: I'm ready...\n", thread\_id);

pthread\_barrier\_wait(&mybarrier);

printf("thread %d: going!\n", thread\_id);

return NULL;

}

int main() {

int i;

pthread\_t ids[THREAD\_COUNT];

int short\_ids[THREAD\_COUNT];

srand(time(NULL));

pthread\_barrier\_init(&mybarrier, NULL, THREAD\_COUNT + 1);

for (i=0; i < THREAD\_COUNT; i++) {

short\_ids[i] = i;

pthread\_create(&ids[i], NULL, threadFn, &short\_ids[i]);

}

printf("main() is ready.\n");

pthread\_barrier\_wait(&mybarrier);

printf("main() is going!\n");

for (i=0; i < THREAD\_COUNT; i++)

pthread\_join(ids[i], NULL);

pthread\_barrier\_destroy(&mybarrier);

return 0;

}

## Adunarea în paralel a n numere

Vom da, ca exemplu de utilizare a thread-urilor, evaluarea în paralel a sumei mai multor numere întregi. Evident, operaţia de adunare a *n* numere, chiar dacă *n* este relativ mare, nu impune cu necesitate însumarea lor în paralel. O facem totuşi pentru că reprezintă un exemplu elocvent de calcul paralel, în care esenţa este reprezentată de organizarea prelucrării paralele, aceeaşi şi pentru calcule mult mai complicate.

Presupunem că se dă un număr natural *n*  şi un vector *a* având componentele întregi *a[0], a[1], . . . a[n-1]*. Ne propunem să calculăm, folosind cât mai multe thread-uri, deci un paralelism cât mai consistent, suma acestor numere. Modelul de paralelism pe care ni-l propunem este ilustrat mai jos, pentru *m* = 8:

Mai întâi sunt calculate, în paralel, următoarele patru adunări:

a[0] = a[0] + a[1]; a[2] = a[2] + a[3]; a[4] = a[4] + a[5]; a[6] = a[6] + a[7];

După ce primele două adunări, respectiv ultimele două adunări s-au terminat, se mai execută în paralel încă două adunări:

a[0] = a[0] + a[2]; a[4] = a[4] + a[6];

In sfârşit, la terminarea acestora, se va executa:

a[0] = a[0] + a[4];

Operaţiile de adunare se desfăşoară în paralel, având grijă ca fiecare adunare să se efectueze numai după ce operanzii au primit deja valori în adunările care trebuie să se desfăşoare înaintea celei curente. Este deci necesară o operaţie de *sincronizare* între iteraţii. Considerând că fiecare operaţie de adunare se execută într-o unitate de timp, din cauza paralelismului s-au consumat doar 3 unităţi de timp în loc de 7 unităţi de timp care s-ar fi consumat în abordarea secvenţială. In calcule s-au folosit 7 thread-uri, din care maximum 4 s-au executat în paralel.

Să considerăm acum problema pentru *n* numere şi să implementăm soluţia, cu intenţia de a folosi un număr maxim de thread-uri. Mai întâi extindem setul de numere până la *m* elemente, unde *m* este cea mai mică putere *l* a lui 2 mai mare sau egală cu *n*, adică: *2l-1 < n <= 2l = m,* unde *l = partea întreagă superioară a lui log2n.* Elementele *a[n], . . ., a[m-1]* vor primi valoarea 0. Determinarea valorii lui **m** (cea mai mică putere a lui 2 mai mare sau egală cu **n**) se face uşor prin:

for (m = 1; n > m; m \*= 2);

Pentru organizarea calculelor în regim multithreading, este convenabil să adoptăm o schemă arborescentă de numerotare a thread-urilor, ilustrată în figura de mai jos pentru 32 de numere.

arbore

Frunzele acestui arbore, reprezentate în pătrăţele, reprezintă **operanzii** de adunat. Nodurile interioare, reprezentate în cerculeţe, reprezintă **threadurile** care efectuează adunările.

Un thread oarecare **i** are doi fii. Threadurile de pe ultimul nivel interior au ca fii câte doi operanzi din tabloul de însumat. Celelalte le vom numi **threaduri interioare** şi au câte două **threaduri fii**, numerotate **2\*i** şi **2\*i+1**. Fiecare thread interior îşi va face propria operaţie de adunare numai după ce cei doi fii ai săi îşi vor termina adunările lor.

Fiecare thread **i** face o adunare de forma *a[s] = a[s] + a[d]*. In cele ce urmează, vom determina indicii **s** şi **d** în funcţie de **i**. Vom numi **threaduri fraţi** threadurile ce se află pe acelaşi nivel, numerotaţi în ordinea crescătoare a vârstei lor. In cazul nostru, 2 şi 3 sunt fraţi cu 2 cel mai mic, 4, 5, 6 şi 7 sunt fraţi cu 4 cel mai mic, 8, 9, . . ., 15 sunt fraţi cu 8 cel mai mic, 16, 17, . . ., 31 sunt fraţi cu 16 cel mai mic ş.a.m.d. Fraţii cei mici de pe fiecare nivel au ca număr o putere a lui doi.

Este uşor de observat că fraţii de pe acelaşi nivel au acelaşi număr de operanzi: dacă **m** este numărul total de operanzi (putere a lui 2), **i** este numărul unui thread de pe un anumit nivel, iar **j** este numărul fratelui cel mic al acestuia, atunci fraţii de pe acest nivel au fiecare câte **m / j** operanzi. Indicele **s** al primului operand al threadului **i** este egal cu suma numărului de operanzi ai fraţilor lui mai mici, iar pentru **d** se mai adaugă jumătate din numărul de operanzi ai threadului, adică **d = s + m / j / 2**.

Determinarea numărului **j** al fratelui cel mic înseamnă găsirea celei mai mari puteri a lui 2 care este mai mică sau egală cu **i** şi ea se determină uşor prin secvenţa:

for (j = m; j > i; j /= 2);

Cu aceste precizări, sursa programului de adunare multithreading este:

#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

}

In legătură cu aşteptările active **while (tid[st] == -1); while (tid[dr] == -1);**

Am folosit acest mod de aşteptare pentru simplitatea codului; mai sunt şi altele posibile:

* Să se pornească threadurile în maniera "bottom up" în arborele asociat.
* variantă "mai puţin ortodoxă" este de a ceda controlat procesorul pentru un interval de timp: **while (tid[st] == -1) sleep(1);**
* Fiecare thread să îşî pornească singur sei doi fii pe care (eventual) îi are.
* Să se folosească un tablou de variabile condiţionale care să semnaleze / să aştepte pornirile threadurilor