Utilizare pthreads; acces exclusiv la locații

Contents

1. 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`

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`

Funcţia de descriere a acţiunii threadului	`void * work(void* a)`

Exemplele care urmează vor clarifica lucrul cu pthreads. Pentru a putea diverse soluţii de rezolvare pentru aceeaşi problemă, vom relua unele dintre problemele prezentate la lucrul cu procese.

2. 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!

3. 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]);
        // pereche.n1 = argv[i]; // Vezi comentariul de la sfârşitul sursei
        // pereche.n2 = argv[i+1]; // Vezi comentariul de la sfârşitul sursei
        // pthread_create(&tid[p], NULL, tippereche, (void*)&pereche); // Vezi
comentariul de la sfârşitul sursei
    }
    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 sunt trei linii comentariu, în care, în loc de `pereche[p]` se foloseşte `pereche`. 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, 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!

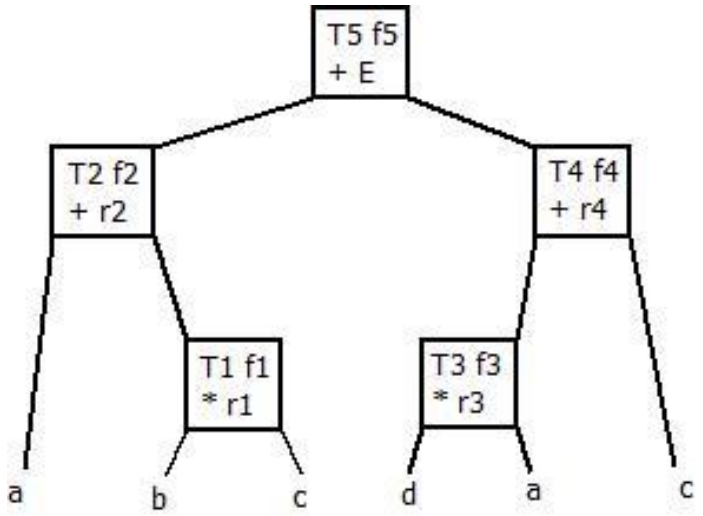
4. Evaluarea expresie aritmetică operator / thread şi paralelizare maximă

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`.

5. 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.

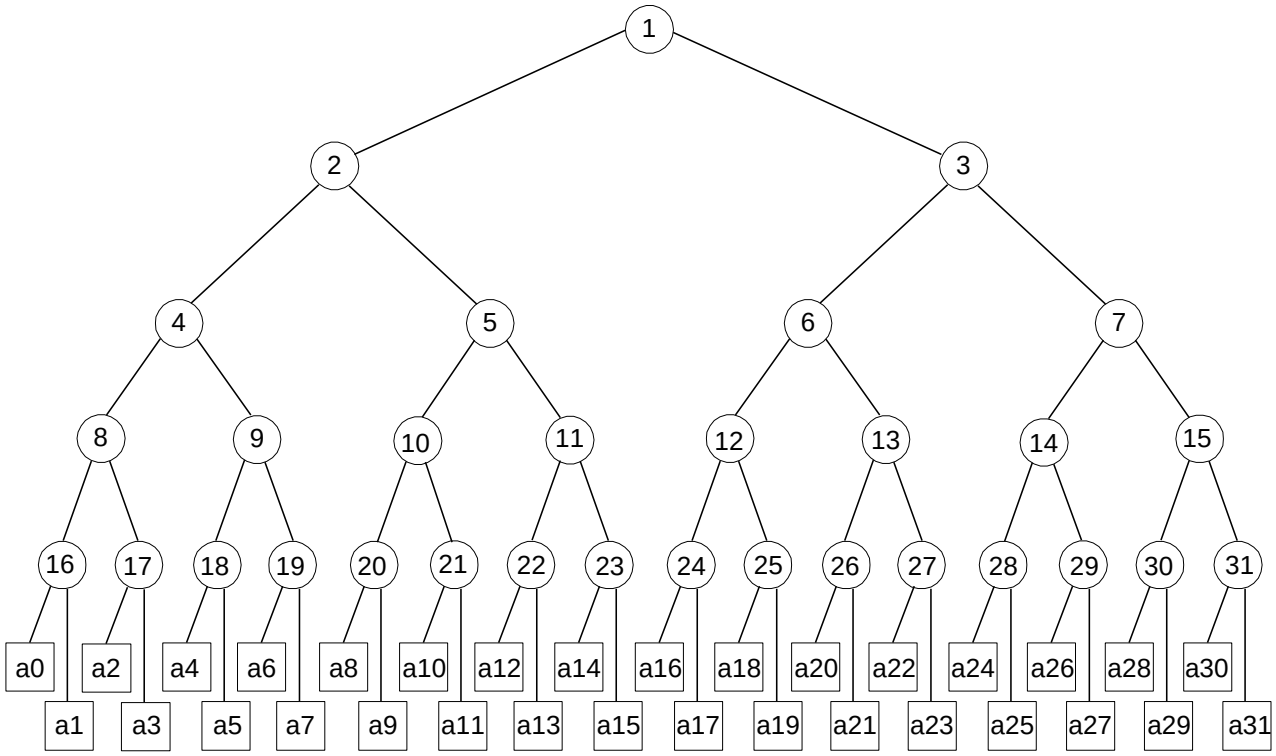
Să considerăm acum problema pentru *n* numere şi să implementăm soluţia, cu intenţia de a folosi un număr maxim de thread-uri. Mai întâi extindem setul de numere până la *m* elemente, unde *m* este cea mai mică

putere *l* a lui 2 mai mare sau egală cu *n*, adică: $2^{l-1} < n <= 2^l = m$, unde *l = partea întreagă superioară a lui* $log_2 n$. 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.



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!
        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 observaţiile de la
exemplul 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
}
```