Laboratorul 1 - Introducere în programarea paralelă cu Pthreads

Responsabili: Radu Ciobanu, Florin Mihalache, Andrei Damian, Delia Stuparu, Dragoș Ușurelu

Programarea paralelă

În mod tradițional, programele pe care le-ați implementat până acum au fost scrise pentru **calcul serial**. Astfel, o problemă care trebuia rezolvată era împărțită într-un set discret de instrucțiuni, care erau executate pe un singur procesor în mod secvențial, una după cealaltă. La un moment dat de timp, o singură instrucțiune putea fi executată.

De cealaltă parte, **programarea paralelă** reprezintă utilizarea simultană a mai multor resurse de calcul pentru a rezolva o problemă. Din punct de vedere al pașilor de rezolvare, problema trebuie întâi împărțită în componente discrete care pot fi rezolvate concurent. Apoi, fiecare astfel de componentă trebuie mai departe împărțită în câte un set de instrucțiuni. În final, pentru a obține paralelism, instrucțiuni ale componentelor diferite ale problemei se pot executa simultan pe procesoare diferite. Pentru a se putea realiza acest lucru, este necesar un mecanism de coordonare a execuției diferitelor componente ale problemei.

Pentru a putea paraleliza în mod eficient o problemă, ea trebuie să poată fi împărțită logic în componente separate care pot fi executate simultan, iar durata execuției paralele a acestor componente trebuie să fie mai mică atunci când avem mai multe resurse de calcul decât atunci când avem o singură astfel de resursă. Pentru a putea executa un program paralel, este necesar să avem fie o mașină de calcul cu mai multe procesoare/core-uri, fie un număr arbitrar de astfel de mașini de calcul conectate printr-o rețea (dar aici extindem conceptul de programare paralelă către **programarea distribuită**).

Atunci când implementăm un program paralel, trebuie să ținem cont de mai multe considerente de design, cum ar fi:

- cum partiționăm problema?
- cum balansăm încărcarea?
- cum realizăm comunicația între componentele care rulează în paralel?
- ce dependențe de date avem?
- cum sincronizăm componentele paralele ale programului nostru?
- cât de mare este efortul de a paraleliza o problemă?

De-a lungul acestui semestru, vom încerca să adresăm cât mai multe din aceste probleme.

Fire de execuție

Un fir de execuție (sau **thread** în engleză) este definit ca un flux independent de instrucțiuni care pot fi planificate de către sistemul de operare. Din punct de vedere al unui programator, un fir de execuție poate fi descris cel mai bine ca o funcție care rulează independent de programul principal, iar un program paralel (cu mai multe fire de execuție, sau **multi-threaded**) poate fi privit ca toată mulțimea de astfel de funcții care pot fi planificate să ruleze simultan și/sau independent de către sistemul de operare.

Atenție! Există o distincție foarte importantă între conceptul de **proces** și cel de **thread**! Veți intra în detaliu la alte materii, dar, în acest moment, este foarte important de reținut faptul că un proces este o instanță de rulare a unui program (și deci **două procese distincte nu partajează spațiul de adrese**, care include stiva de program, variabile, date, etc.), pe când un thread este o unitate de lucru a unui proces (deci **mai multe thread-uri pot avea acces partajat la variabile și alte date**).

Datorită faptului ca firele de execuție ale aceluiași proces partajează resurse, modificările făcute de către un thread asupra acelor resurse (cum ar fi, de exemplu, închiderea unui fișier) vor fi observate de toate thread-urile acelui proces. Mai mult, doi pointeri cu aceeași valoare referă aceleași date, iar scrierea și citirea în/din aceeași zonă de memorie este posibilă, dar necesită sincronizare explicită din partea programatorului (mai multe detalii despre ce înseamnă sincronizarea și cum se realizează veți afla în laboratorul 2).

În general, programele care pot beneficia de implementare multi-threaded au câteva trăsături comune:

- conțin componente computaționale care se pot executa în paralel
- au date pe care se poate opera în paralel
- se blochează ocazional așteptând după I/O
- trebuie să răspundă la evenimente asincrone
- anumite componente de execuție au o prioritate mai mare decât altele.

Pthreads (thread-uri POSIX)

La începutul programării paralele, producătorii de hardware își implementau propriile versiuni de thread-uri, care difereau considerabil între ele, ceea ce ducea la dificultăți în dezvoltarea de aplicații multi-threaded portabile. Din acest motiv, o interfață standardizată de programare multi-thread a fost necesară, iar acest lucru a fost specificat, pentru sisteme UNIX, de către standardul IEEE POSIX 1003.1c în anul 1995. Astfel, implementările de thread-uri care aderă la acest standard sunt denumite thread-uri POSIX, sau Pthreads, iar astăzi majoritatea furnizorilor de hardware oferă și Pthreads.

Din punct de vedere al unui programator, Pthreads sunt definite ca o mulțime de tipuri și funcții pentru limbajul C, implementate într-un header numit **pthread.h** și o bibliotecă numită **pthread** (deși, în unele implementări de Pthreads, această bibliotecă poate fi inclusă într-o altă bibliotecă).

Implementarea unui program paralel folosind Pthreads

Compilare și rulare

Pentru a compila un program care folosește Pthreads, va trebui să link-ăm biblioteca de Pthreads. Rularea programului se realizează ca pentru orice alt program C. De exemplu, dacă folosim compilatorul GCC, putem compila și rula un program Pthreads astfel:

```
gcc -o program program.c -lpthread
./program
```

Atenție! Ca să implementăm un program care folosește Pthreads, trebuie să includem header-ul **pthread.h**.

Crearea și terminarea thread-urilor

Într-un program C cu Pthreads, inițial există un singur fir de execuție, numit thread-ul principal (**main thread**). Oricare alt fir de execuție trebuie creat și pornit în mod explicit de către programator, acest lucru făcându-se prin intermediul funcției **pthread_create**, care poate fi apelată de oricâte ori și de oriunde din cod. Funcția are următoarea semnătură:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
```

Parametrul *thread* (de tip *pthread_t*) reprezintă un identificator pentru noul thread returnat de această funcție, ce poate fi apoi folosit pentru a opri sau a referi thread-ul. Parametrul *attr* este utilizat pentru a seta diferite atribute pentru firul de execuție care se creează, și poate fi pus pe NULL dacă se dorește păstrarea valorilor implicite. *start_routine* este un pointer la funcția care va fi executată pe firul de execuție nou-creat la pornirea sa. În final, prin intermediul parametrului *arg*, putem trimite un singur parametru către funcția de thread, care trebuie pasat prin referință ca un pointer de tipul *void* (dacă nu se dorește trimiterea unui parametru, putem lăsa NULL). Funcția returnează 0 dacă thread-ul nou s-a creat și s-a pornit cu succes, sau un cod de eroare în caz contrar.

Un thread creat astfel poate la rândul lui crea alte thread-uri, neexistând o ierarhie sau dependență între ele. Numărul maxim de thread-uri care poate fi creat de un proces depinde de implementarea Pthreads, dar în general nu se recomandă să avem un număr de fire de execuție mai mare decât numărul de core-uri de pe mașina pe care rulăm, din motive de overhead (aceasta este o discuție mai lungă pe care vă recomandăm să o purtați cu asistentul vostru).

Funcția pe care un thread o execută atunci când apelăm **pthread_create** trebuie să aibă următoarea semnătură:

```
void *f(void *arg) {
    [...]

    return NULL;
    /*
     *  In urma ieșirii din funcție, se va apela implicit pthread_exit(NULL);
     */
}
```

Parametrul *arg* al funcției de mai sus este primit de funcția **f** la execuția ei pe un thread nou atunci când se apelează **pthread_create**, fiind echivalentului parametrului *arg* de la **pthread_create**. Limitarea dată de faptul că putem trimite un singur parametru funcției de thread poate fi rezolvată prin crearea unei structuri cu oricâți membri, pe care o putem apoi da ca parametru prin referință. Funcția **pthread_exit** poate fi apelată pentru a termina thread-ul care o apelează, și primește ca parametru fie NULL, fie o valoare de retur care va fi pasată mai departe. Apelul funcției **pthread_exit** nu este obligatoriu, pentru că ea este apelata implicit la ieșirea din subrutină, iar parametrul acesteia va fi egal cu valoarea de retur a funcției.

În momentul în care apelăm **pthread_create**, firul de execuție nou-creat va rula în paralel cu thread-ul principal. Acest lucru înseamnă că tot codul de după apelul funcției **pthread_create** se va executa în paralel cu codul noului thread. Dacă dorim ca un fir de execuție să aștepte terminarea unui alt thread, putem realiza acest lucru prin funcția **pthread_join**. Prin apelul acestei funcții, ne putem asigura că thread-ul apelant se blochează până când celălalt fir de execuție își termină procesarea (adică termină de executat funcția de thread). Funcția **pthread_join** are următoarea semnătură:

```
int pthread_join(pthread_t thread, void **retval);
```

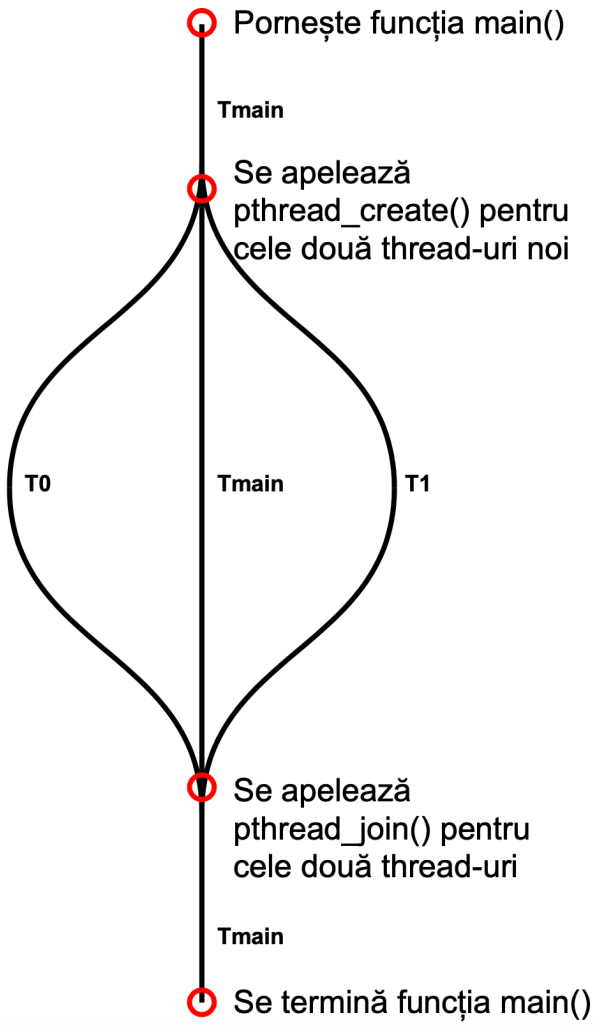
Parametrul *thread* reprezintă identificatorul thread-ului pe care îl așteptăm, iar *retval* reprezintă valoarea de retur a funcției thread-ului așteptat (și poate fi pus pe NULL dacă nu avem nevoie de această informație). Funcția returnează 0 în caz de succes sau un cod de eroare în caz contrar.

Un exemplu complet de program C care folosește Pthreads se poate observa mai jos. În acest exemplu, se creează două fire de execuție care vor rula aceeași funcție **f**, dar fiecare din ele va primi ca parametru ID-ul curent (0 sau 1). După ce pornește cele două fire de execuție noi, thread-ul principal le așteaptă pe ambele să termine, după care iese și el.

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

#define NUM_THREADS 2

void *f(void *arg)
{
    long id = *(long*) arg;
    printf("Hello World din thread-ul %ld!\n", id);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int r;
    long id;
    void *status;
    long arguments[NUM_THREADS];

    for (id = 0; id < NUM_THREADS; id++) {
        arguments[id] = id;
        r = pthread_create(&threads[id], NULL, f, (void *) &arguments[id]);

        if (r) {
            printf("Eroare la crearea thread-ului %ld\n", id);
            exit(-1);
        }
    }
```

```
    for (id = 0; id < NUM_THREADS; id++) {
        r = pthread_join(threads[id], &status);

        if (r) {
            printf("Eroare la asteptarea thread-ului %ld\n", id);
            exit(-1);
        }
    }

    return 0;
}
```

O reprezentare vizuală a execuției programului de mai sus poate fi observată în figura de mai jos.



În figura de deasupra, se poate observa că inițial există un singur fir de execuție, cel principal (marcat cu **Tmain**). În momentul în care acesta apelează **pthread_create** de două ori, el creează și pornește alte două fire de execuție noi (**T0** și **T1**) care vor rula funcția **f** cu parametri diferiți (0, respectiv 1). În acest moment, avem trei thread-uri care rulează în paralel. Când thread-ul principal apelează **pthread_join**, acesta se blochează până când cele două fire de execuție secundare se termină (adică până când termină de executat funcția **f**). Dacă unul din thread-urile secundare și-a terminat execuția înainte de apelul **pthread_join**, funcția va returna instant. La finalul programului, thread-ul principal se va opri și el.

Atenție! Găsiți aici un „cheat sheet” cu cele mai importante elemente Pthreads.

Exerciții

1. Compilați și executați codul din fișierul **example.c** din scheletul de laborator, pe care îl găsiți pe acest repository de GitHub [https://github.com/APD-UPB/APD], pe care trebuie să îl clonați, folosind comanda `git clone`. Schimbați numărul de thread-uri și observați cum se schimbă comportamentul programului.
2. Schimbați numărul de thread-uri din cod astfel încât să fie egal cu numărul de core-uri de pe mașina pe care rulați, astfel încât, rulând codul pe un alt calculator, numărul de thread-uri să se schimbe automat. Verificați **Hint 1** de mai jos pentru informații suplimentare.
3. Modificați funcția **f** astfel încât mesajul „Hello World” să fie afișat iterativ de 100 de ori de fiecare thread, împreună cu indicele iterației. Întrebare: codul afișează mesajele în ordinea așteptată de voi?
4. Modificați programul astfel încât să creeze două thread-uri, fiecare thread rulând propria sa funcție.
5. Pornind de la codul din fișierul **add_serial.c** din arhiva de laborator, paralelizați incrementarea elementelor unui vector cu 100. Acest lucru va presupune împărțirea iterației de adunare la toate thread-urile într-un mod cât mai echitabil. Verificați **Hint 2** de mai jos pentru informații suplimentare.
6. Demonstrați că programul vostru scalează (adică durează mai puțin când rulați cu mai multe thread-uri). Verificați **Hint 3** de mai jos pentru informații suplimentare.

Hint 1

Pentru a putea obține numărul de core-uri de pe un calculator, se poate folosi funcția **sysconf** astfel:

```
#include <unistd.h>

long cores = sysconf(_SC_NPROCESSORS_CONF);
```

Hint 2

Pentru exercițiul 5, avem un vector de **N** elemente pe care vrem să-l împărțim în mod (aproximativ) egal la **P** thread-uri, unde fiecare thread are un **ID** de la 0 la P-1. Astfel, fiecare thread va itera pe câte o secțiune din vectorul inițial, fără a afecta operațiile celorlalte thread-uri. Este necesar deci să calculăm indexul de **start** și indexul de final (**end**) pentru fiecare thread. Un mod de a calcula aceste două valori poate fi următorul:

```
int start = ID * (double)N / P;
int end = min((ID + 1) * (double)N / P, N);
```

Hint 3

Pentru a putea observa mai bine scalabilitatea unui program, este necesar ca acesta să dureze măcar câteva secunde, deoarece, în caz contrar, timpul de inițializare, alte programe care rulează pe calculator și overhead-ul cauzat de planificarea firelor de execuție ar putea afecta timpii de execuție suficient cât să nu vedem scalabilitatea doar prin măsurarea timpului total de execuție.

Mai mult, inițializarea serială a vectorului (în main) are o durată comparabilă cu execuția pe un thread a operației de paralelizat. De aceea, **pentru exercițiul 5, se recomandă să creșteți durata de execuție a unui thread prin repetarea iterativă a operațiilor realizate în funcția de thread**.

Ca să verificați dacă un program scalează, trebuie să îi măsurați durata de execuție atât la o rulare secvențială (cu un singur thread), cât și la o rulare cu mai multe fire de execuție. Pentru acest lucru, puteți folosi comanda **time** în linia de comandă, astfel:

```
$ time ./program
real    0m6.958s
user    0m6.745s
sys     0m0.010s
```

Timpul care ne interesează este cel real (adică timpul „pe ceas”).

Hint 4

O metodă bună de a face debugging la un program C multi-threaded este utiliarul **gdb**. Pe lângă comenzile **gdb** pe

care le știți deja, ar mai fi de interes comenzile **info threads** (care afișează informații despre thread-urile existente la momentul curent de timp) și **thread <N>** (care mută contextul de execuție pe thread-ul N).

apd/laboratoare/01.txt · Last modified: 2021/10/22 15:56 by florin.mihalache