Tutoriat 3 (Operating Systems)
Fire de executie (threads)

Contents

1 Introducere

Un **thread (fir de executie)**, cunoscut si sub numele de **lightweight process**, este cea mai mica unitate de procesare ce poate fi programata spre executie (un flux de instructiuni din cadrul unui program). Initial, un proces este format dintr-un singur fir de executie. Procesul poate fi impartit in mai multe fire, fiecare executandu-si blocul de instructiuni asociat. Asemanari si deosebiri relevante intre fire si procese:

- Un proces este un program in executie; un fir este o parte dintr-un proces.
- Orice proces are asociat un **PCB**, in timp ce orice fir are un **TCB (thread control block)**, ce contine ID-ul firului, program counter-ul, o multime de registri, si o stiva (pentru *activation records*).
- Firele si procesele trec prin aceleasi stare (new, ready, running, blocked, terminated).
- Procesele noi sunt **forked**, iar firele noi sunt **create**. Ambele pot indeplini **context switch-uri**, dar crearea proceselor si context switching-ul intre ele dureaza mai mult decat la fire. Un *proces parinte* asteapta finalizarea executiei copiilor prin intermediul apelului **wait**, in timp ce un fir poate astepta finalizarea executiei altui fir prin intermediul apelului **join** (totusi, nu exista o relatie *parinte-copil* la fire cum exista la procese).
- Procesele comunica intre ele prin intermediul anumitor mecanisme mai complexe (*memorie partajata* si *message passing*), in timp ce firele aceluiasi proces au acces la **codul procesului**, **sectiunea de date**, **sectiunea heap**, si alte resurse (precum fisierele deschise sau semnale). Astfel, comunicarea intre fire este mai usoara si eficienta.

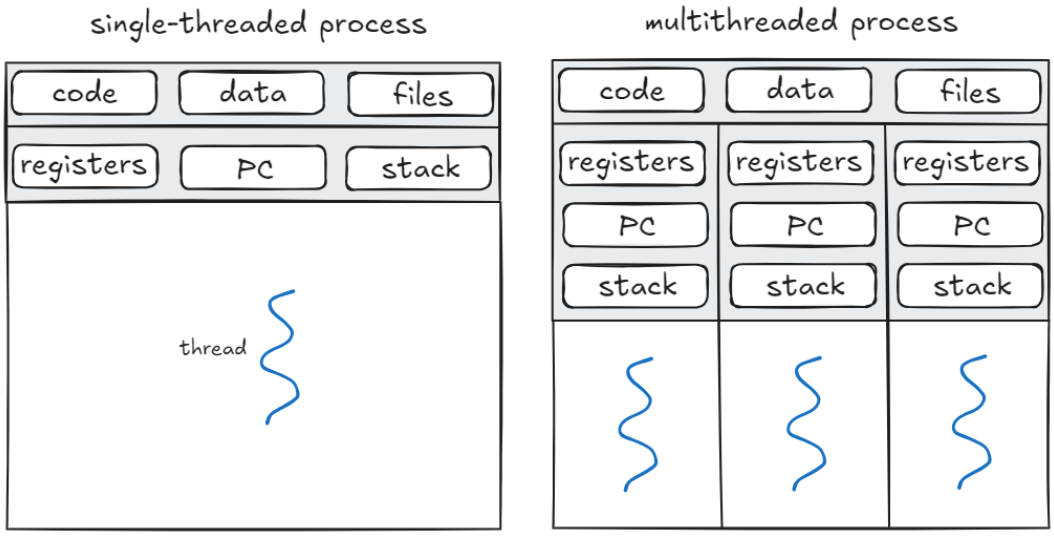

Figure 1: Un proces single-threaded si unul multithreaded

Beneficiile unei aplicatii multithreaded (mai tarziu vom vorbi si despre dificultatile intampinate):

- **Responsiveness:** daca o parte dintr-o aplicatie multithreaded se blocheaza sau executa o operatie intensiva din punct de vedere al resurselor, restul aplicatiei ramane functionala. In schimb, o aplicatie single-threaded s-ar fi blocat complet.
- **Resource sharing:** procesele pot comunica intre ele numai prin intermediul anumitor mecanisme (memorie partajata, message passing). Pe de alta parte, firele aceluiasi proces partajeaza in mod implicit resursele procesului, oferind comunicare mai rapida.
- **Economy:** crearea proceselor si context switch-urile sunt destul de costisitoare ca si timp. Cum thread-urile sunt mai mici si implicit isi partajeaza anumite resurse, timpii acestor operatii sunt mai acceptabili.
- **Scalability:** firele pot rula in paralel pe mai multe nuclee.

2 Programarea multicore

In crearea unui sistem multicore, pot fi intampinate urmatoarele dificultati:

- **Identificarea sarcinilor:** trebuie examinata aplicatia pentru a putea determina o impartire a sarcinilor. Ar fi ideal ca oricare doua sarcini sa fie independente una de cealalta, pentru a permite rularea in paralel pe mai multe nuclee.
- **Echilibru:** in identificarea sarcinilor, acestea ar trebui sa fie aproximativ la fel din punct de vedere al muncii totale efectuate. Nu prea merita folosirea unui nucleu pentru rularea unui fir care nu contribuie intr-un mod semnificativ la sarcina generala.
- **Impartirea datelor:** datele accesate si manipulate de catre fire trebuie sa fie impartite pe mai multe nuclee.
- **Dependente intre date:** datele accesate trebuie sa fie examinate pentru dependente intre fire. In cazul in care este observata o dependenta, executia sarcinilor trebuie sa fie sincronizata intr-un mod adecvat (vom discuta pe viitor despre sincronizarea proceselor).
- **Testare si debugging:** cand un program ruleaza in paralel pe mai multe nuclee, exista mai multe drumuri de executie. Este mai complicat sa testezi si sa faci debugging la o astfel de aplicatie.

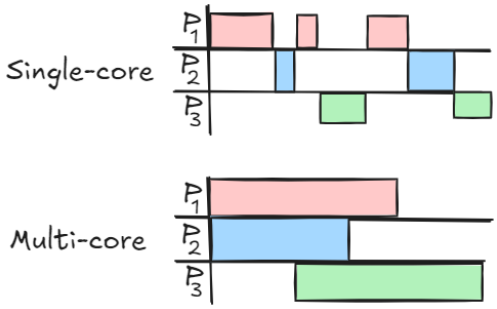

Figure 2: Executie concurenta pe un sistem single-core si paralelism pe un sistem multi-core

Reamintim urmatoarele doua proprietati: **concurenta** este abilitatea unui sistem de a executa mai multe sarcini in acelasi timp, alternand intre ele intr-un mod aproape instant; **paralelismul** reprezinta executia cu adevarat simultana a sarcinilor respective. Exista doua tipuri de paralelism:

- **Paralelismul datelor:** distribuirea unor submultimi ale aceluiasi set de date peste mai multe nuclee, fiecare nucleu executand aceleasi operatii pe datele respective.
- **Paralelismul sarcinilor:** distribuirea firelor peste mai multe nuclee, fiecare fir executandu-si propriul flux unic de instructiuni.

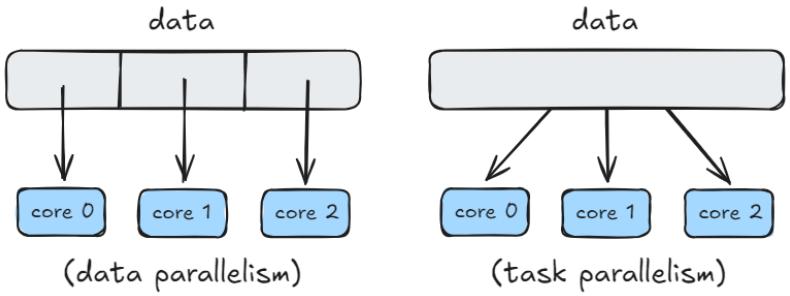

Figure 3: Paralelismul datelor si paralelismul sarcinilor

Presupunem ca avem o aplicatie, atat cu **componente seriale** (se executa secvential, fara paralelism), cat si cu **componente paralele**. **Legea lui Amdahl** este o formula care prezice cat de mult se poate eficientiza o sarcina,

cand doar o portiune (cea paralela) din sarcina respectiva poate fi imbunatatita, prin adaugarea mai multor nuclee de calcul. Legea sustine faptul ca performanta aplicatiei este limitata de partea seriala, indiferent de cat de mult este imbunatatita partea paralela. Daca notam cu **S** portiunea seriala a aplicatiei, care trebuie efectuata pe un sistem cu **N** nuclee de calcul, formula este urmatoarea:

$$\texttt{speedup} \leq \frac{1}{S + \frac{1-S}{N}}$$

De exemplu, pentru o aplicatie care este 75% paralela (deci 25% seriala), eficientizarea maxima care poate fi obtinuta cu o infinitate de nuclee de calcul ar fi marginita superior astfel:

$$\texttt{speedup} \leq \frac{1}{0.25 + \frac{1-0.25}{\inf}} = \frac{1}{0.25} = 4$$

Astfel, sarcina poate fi realizata maxim de 4 ori mai rapid. Daca am folosi patru procesoare, am obtine:

$$\texttt{speedup} = \frac{1}{0.25 + \frac{1-0.25}{4}} \approx 2.28$$

2.1 Modele de multithreading

Un **fir de executie de utilizator (user thread)** este un fir gestionat total de catre utilizator in user space, fara implicarea kernel-ului (kernel-ul nu stie de existenta firului respectiv). In consecinta, daca un fir de utilizator se blocheaza, atunci se blocheaza tot procesul, deoarece kernel-ul blocheaza la nivel de proces, fara a avea idee despre firele procesului. Un **fir de executie de kernel (kernel thread)** este un fir gestionat in mod direct de catre kernel, care ii poate planifica executia pe CPU si poate gestiona blocarea firului.

Firele de utilizator sunt mai rapide din punct de vedere al timpului necesar crearii, sincronizarii, si realizarii context switch-urilor; totusi, user thread-urile nu sunt capabile de paralelism daca nu sunt mapate la kernel thread-uri, deoarece numai kernel-ul poate aloca timp de executie pe CPU si poate gestiona cererile I/O. Din acest motiv, este necesara o **mapare** intre user si kernel threads.

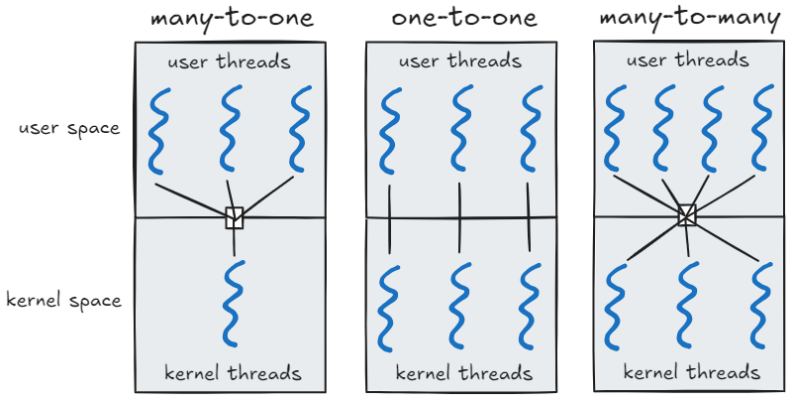

Figure 4: Modele pentru maparea firelor de utilizator la fire de kernel

2.1.1 Modelul many-to-one

Cu acest model, mai multe user thread-uri sunt mapate la un singur kernel thread. Gestionarea firelor este eficienta, fiind realizata de catre biblioteca utilizatorului in user space (firele de kernel fiind mai lente de creat si alternat, din cauza implicarii kernel-ului). Deoarece doar un singur fir de utilizator poate accesa kernel-ul la un moment dat (user thread-urile avand acces la numai un singur fir de kernel), acest model nu poate furniza paralelism. De asemenea, daca un user thread se blocheaza, se va bloca si firul de kernel; in consecinta, se blocheaza tot procesul.

2.1.2 Modelul one-to-one

Fiecare fir de utilizator este mapat la un fir de kernel; daca un user thread se blocheaza, altul se poate executa. De asemenea, pe un sistem multithreaded, se poate practica paralelismul. Dezavantajul este timpul petrecut creand un numar ridicat de fire de kernel, incetinind performanta sistemului.

2.1.3 Modelul many-to-many

Mai multe user thread-uri sunt mapate la un numar mai mic sau egal de kernel thread-uri. Se pot folosi mai multe nuclee pentru a rula mai multe fire in paralel. Daca un user thread se blocheaza, se poate executa alt user thread pe un kernel thread diferit, prevenind blocarea intregului proces. Dezavantajele ar fi timpul necesar crearii si gestionarii kernel thread-urilor, iar complexitatea modelului este mai ridicata decat la celelalte doua, dar este o abordare flexibila ce ofera concurenta si paralelism.

3 Implicit threading

Proiectarea unei aplicatii cu sute sau mii de fire de executie este o sarcina dificila. Un mod de a adresa problemele care pot aparea este ca responsabilitatea gestionarii si crearii firelor sa fie transferata de la programator la diverse compilatoare si biblioteci. Aceasta strategie poarta numele de **implicit threading**.

3.1 Thread pools

In lucrul cu fire, un dezavantaj ar fi timpul necesar crearii unui fir; de asemenea, stergerea si dealocarea resurselor unui fir odata ce si-a finalizat sarcina reprezinta o pierdere de timp si de resurse. O solutie ar fi utilizarea unui **thread pool**: o colectie de fire, inactive, deja create, care asteapta sa le fie atribuite sarcini.

In loc sa fie creat un fir nou de fiecare data cand trebuie indeplinita o sarcina, thread pool-ul selecteaza un fir inactiv si ii atribuie sarcina respectiva; odata ce isi finalizeaza executia, firul devine din nou inactiv, asteptand urmatoarea sarcina. Daca nu exista vreun fir valabil la momentul respectiv, sarcina este plasata intr-o coada, pana cand se elibereaza un fir. Aceasta abordare prezinta urmatoarele avantaje:

- Gestionarea unei sarcini folosind un fir deja existent este, de obicei, mai rapid decat crearea unui fir nou.
- Numarul de fire care pot exista la un moment dat este limitat. Acesta este un detaliu important pentru sistemele care nu pot sustine un numar ridicat de fire.
- Separarea sarcinii in sine de catre logica crearii si executiei sarcinii respective ofera mai multa flexibilitate.

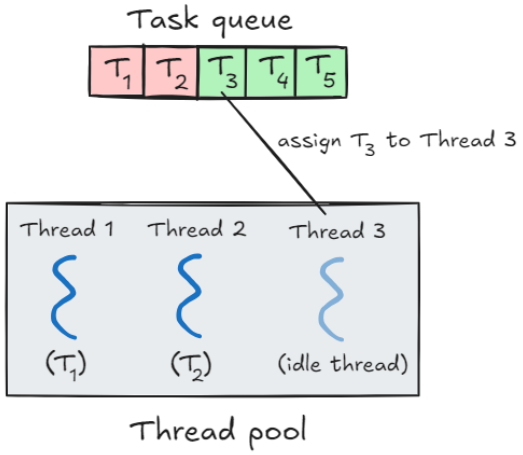

Figure 5: O coada de sarcini si un thread pool

3.2 Fork join

Este utilizat un framework care se ocupa in totalitate de crearea si planificarea firelor. Astfel, programatorul lucreaza cu conceptul de sarcini, nu cu gestionarea firelor. Aceasta metoda este impartita in doua etape:

- **Fork (impartirea firelor):** este identificata o sarcina dificila (intensiva computational). Un fir incepe executia sarcinii. Daca este prea mare pentru a putea fi gestionata de catre un singur fir, atunci se realizeaza **fork-uri** in mod recursiv, spargandu-se in sarcini mai mici.

- **Join (colectarea rezultatelor):** cand firul principal are nevoie de rezultatul altui fir, apeleaza functia **join**. Rezultatele sarcinilor mai mici sunt combinate pentru a furniza rezultatul final.

3.3 OpenMP

OpenMP este un API pentru programe scrise in C, C++ si FORTRAN, care permite introducerea *paralelismului* cu *memorie partajata*. Sunt identificate si marcate **regiunile paralele** (sectiunile de cod care ar putea rula in paralel). Initial, programul este reprezentat de un singur fir de executie; cand OpenMP detecteaza o regiune paralela, firul respectiv creeaza (prin intermediul apelului *fork*) niste fire noi care sa execute acea sectiune de cod in paralel, ca apoi in final sa fie apelat **join** pentru toate firele.

Datoria programatorului este de a identifica regiunile paralele si de a insera instructiuni speciale ca si marcaje pentru OpenMP, indicand faptul ca regiunea respectiva trebuie executata in paralel. In plus, programatorul are libertatea de a modifica diversi parametri, cum ar fi numarul de fire create pentru o asemenea regiune.

3.4 Grand Central Dispatch

Grand Central Dispatch (GCD) este o tehnologie dezvoltata de catre Apple pentru sistemele macOS si iOS. GCD permite programatorilor sa identifice sectiuni de cod (sarcini) ce pot rula in paralel. Cand o sarcina trebuie sa fie efectuata, aceasta este adaugata intr-o coada (**dispatch queue**), care poate fi de doua feluri:

- **Serial (main) queue:** sarcinile sunt executate una dupa alta in ordine FIFO (dar nu in paralel). De asemenea, programatorul poate crea mai multe asemenea cozi.
- **Concurrent queue:** sarcinile sunt incepute in ordine FIFO, dar pot rula in paralel.

3.5 Intel Thread Building Blocks (TBB)

Este o biblioteca pentru proiectarea aplicatiilor cu componente paralele in C++. Programatorul identifica sarcinile ce pot rula in paralel, iar **task scheduler-ul** le mapeaza la fire.

4 Probleme legate de threading

4.1 fork() and exec() syscalls

Cand un fir dintr-un proces apeleaza **fork()**, noul proces va duplica cumva toate firele, sau va fi single-threaded? Unele sisteme UNIX au ales sa aiba doua versiuni pentru *fork*: una care copiaza toate firele, si alta care copiaza doar firul apelant. Daca un fir invoca apelul **exec()**, programul specificat ca si parametru va inlocui intregul proces (implicit si toate firele).

4.2 Signal handling

Un **semnal** are scopul de a informa un proces ca a avut loc un anumit eveniment. Orice semnal urmeaza tiparul de mai jos:

- Un semnal este generat de catre un anumit eveniment.
- Semnalul este trimis catre un proces.
- Odata receptionat, semnalul trebuie gestionat.

Orice semnal are un **default signal handler** rulat de catre kernel pentru gestionarea semnalului respectiv. Aceasta rutina poate fi suprascrisa de catre un **user-defined signal handler** care are acelasi scop: de a gestiona semnalul respectiv. Pentru programele single-threaded este simplu: semnalele sunt livrate procesului. Totusi, intr-un program multithreaded, unde ar trebui sa fie trimis semnalul respectiv? Avem urmatoarele optiuni:

- Semnalul este transmis firului pentru care se aplica semnalul respectiv.
- Semnalul este transmis catre fiecare fir al procesului.
- Semnalul este transis numai catre anumite fire ale procesului.
- Un anumit fir primeste responsabilitatea de a gestiona toate semnalele procesului.

4.3 Anularea unui fir

Anularea unui fir se refera la terminarea unui fir inainte ca acesta sa isi finalizeze executia. De exemplu, cand mai multe fire cauta ceva intr-o baza de date si unul din fire gaseste si returneaza rezultatul, restul firelor pot fi anulate. O problema cu anularea firelor apare atunci cand target thread-ul manipuleaza date partajate cu alte fire. Un fir anulat se mai numeste si **target thread**. Anularea se poate produce in doua feluri:

- **Asincron (asynchronous cancellation):** un fir anuleaza target thread-ul cat mai rapid.
- **Amanat (deferred cancellation):** un fir indica faptul ca target thread-ul ar trebui sa fie anulat, iar target thread-ul verifica in mod periodic daca ar trebui sa fie anulat sau nu; in cazul in care se doreste anularea sa, aceasta va fi indeplinita cand va ajunge la un **cancelation point**.

4.4 Thread-local storage

Firele aceluiasi proces partajeaza datele procesului. Totusi, exista anumite situatii in care dorim ca fiecare fir sa aiba propria sa copie a datelor; un **thread-local storage (TLS)**. Acesta difera de variabilele locale, deoarece acestea sunt vizibile numai pentru un singur apel de functie, in timp ce TLS-ul este vizibil pentru multiple apeluri. TLS-ul este unic pentru fiecare fir in parte.

4.5 Scheduler activities

O problema ar mai fi comunicarea dintre kernel si biblioteca utilizatorului pentru gestionarea firelor. Multe OS-uri plaseaza o structura de date intermediara intre maparea dintre user si kernel thread-uri, cunoscuta sub numele de **lightweight process (LWP)**; acesta este ca un fel de procesor virtual pe care poate rula un user thread. Fiecare LWP este atasat de un kernel thread.

Exista un model de comunicare intre user si kernel thread-uri numit **scheduler activation**. Kernel-ul furnizeaza aplicatia cu o multime de procesoare virtuale (LWPs), pe care pot rula firele. De asemenea, kernel-ul are responsabilitatea de a semnala anumite evenimente aplicatiei; aceasta procedura poarta numele de **upcall**. Upcall-urile sunt gestionate de catre biblioteca utilizatorului care se ocupa de fire, prin intermediul unui **upcall handler**.