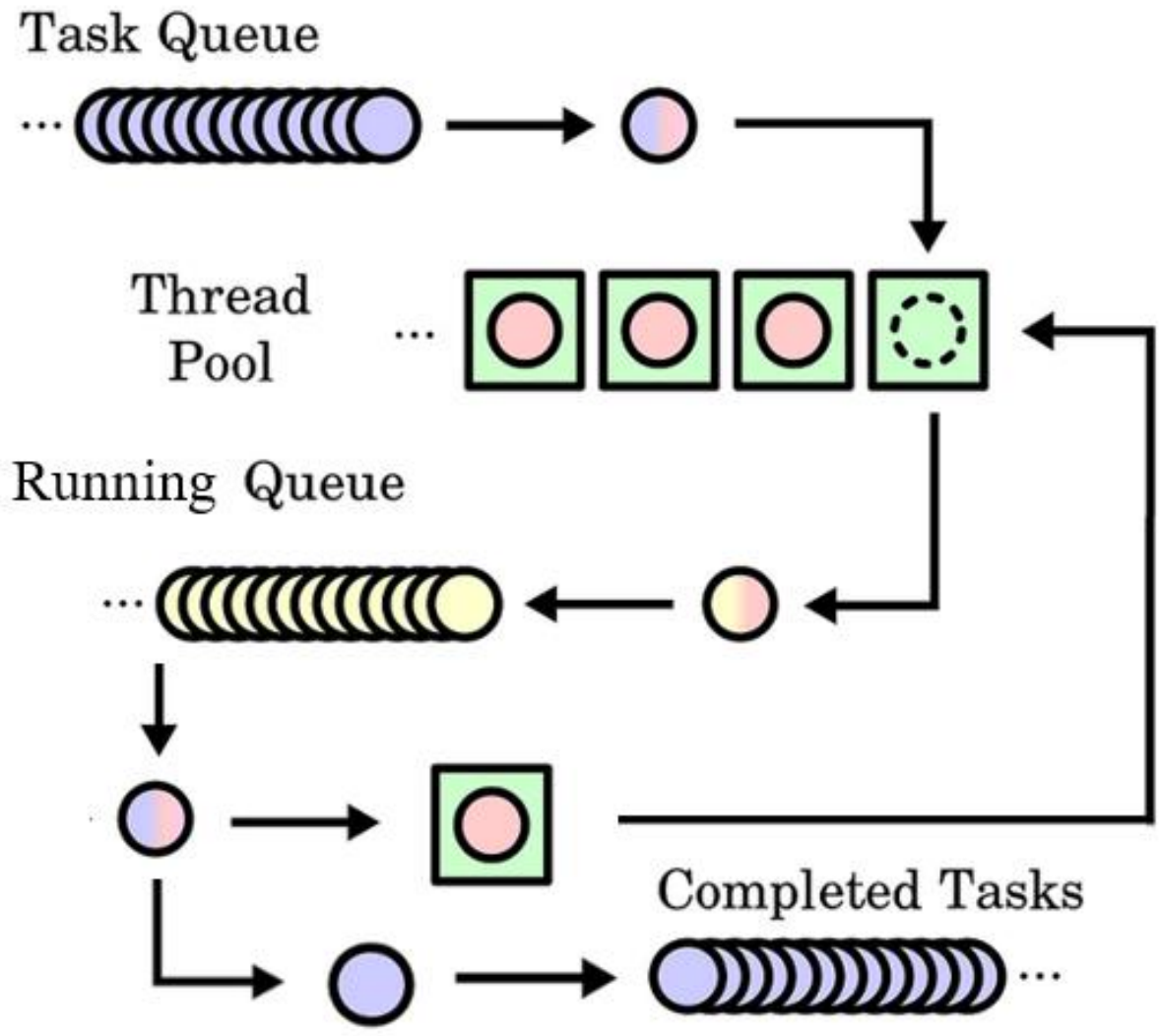
Structuri de date și Algoritmi
(Seria CB)
Tema 2 - ThreadPoolExecutor

Responsabili temă	Maria-Anca Bălușoiu, Miruna Chiricu, Roxana Știucă
Data publicării	14.04.2022
Termen predare	05.05.2022 Se acceptă teme trimise cu penalizare de 10 puncte/zi (din maxim 150 de puncte) până la data de 08.05.2022 (ora 23:59)
Versiune document	1.0

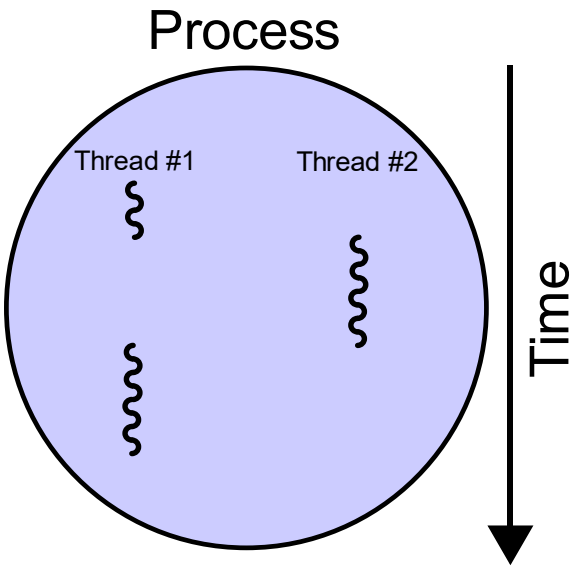


1. Introducere

Conceptul de thread (fir de execuție) definește cea mai mică unitate de procesare ce poate fi programată spre execuție de către sistemul de operare. Este folosit în programare pentru a eficientiza execuția programelor, executând porțiuni distincte de cod în paralel în interiorul aceluiași proces. Acest lucru este posibil datorită CPU-urilor: Fiecare nucleu (core) dintr-un CPU poate avea două fire de execuție (poate rula două sarcini simultan).

Ex : Un procesor cu 8 nuclee → 16 fire de execuție.

ThreadPoolExecutor este un planificator de thread-uri care primește task-uri de rezolvat și le alocă thread-urilor available pentru a le lansa în execuție. Practic, facilitează/eficientizează rularea task-urilor în paralel.



2. Cerință

Scopul temei este simularea unui planificator al firelor de execuție al unui procesor. Considerați că aveți maxim **N thread-uri** (fire de execuție) care pot executa **task-uri** (sarcini) în paralel.

Vom caracteriza o structură **Thread** prin următoarele caracteristici:

- **ID,** identificatorul unic al thread-ului în cadrul planificatorului:
  - este atribuit automat la crearea **pool**-ului de thread-uri de către planificator.

Vom caracteriza o structură **Task** prin următoarele caracteristici:

- **ID**, identificatorul unic al task-ului în cadrul planificatorului:
  - este atribuit automat la crearea task-ului de către planificator. Valoarea lui este un număr natural cuprins între [1 - 32767]
  - este atribuit ca fiind cel mai mic număr disponibil
  - dacă un task a avut ID-ul X iar acesta și-a terminat execuția, ID-ul X devine disponibil

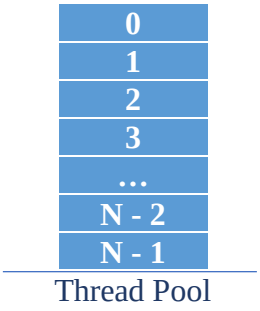
- **Thread**, firul de execuție alocat rezolvării task-ului.
    - va fi alocat automat de către planificator
- **prioritate**, este un număr natural cuprins intre [0 - 127]
    - trimisă ca parametru la crearea task-ului.
- **timp de execuție**, număr natural pozitiv reprezentabil pe 32 de biți
    - trimis ca parametru la crearea task-ului

3. Implementare

Planificatorul nostru de thread-uri (ThreadPoolExecutor) va dispune de o stivă de thread-uri și 3 cozi de task-uri, astfel:

1. **Stiva de thread-uri :**

Inițial, creăm un **pool** de **N thread**-uri, atribuind ID-uri în ordine descrescătoare (de la N - 1 la 0) și le adăugam într-o stivă. Această stivă va reprezenta thread-urile available – care așteaptă să primească un task. Stiva rezultată va arăta astfel:



2. **Cozile de task-uri:**

a) coada waiting

Task-urile se află în așteptarea preluării lor de către un thread.

- Task-urile din această coadă sunt ordonate astfel:
    - descrescător după prioritate
    - crescător după timp de execuție rămas, în caz de priorități egale
    - crescător după ID, în caz de prioritate și timp de execuție rămas egale.

b) coada running

Task-urile rulează în paralel pe procesoare, în cadrul unor thread-uri, cât timp au nevoie pentru a se termina (până când timpul lor de execuție ajunge la 0). Aceste task-uri ce rulează în paralel se vor implementa printr-o coadă. Această coadă va avea max N elemente.

- Task-urile din această coadă sunt ordonate la fel ca în coada waiting.

c) coada finished
Task-urile s-au terminat.

- Task-urile din această coadă sunt ordonate astfel:
    - task-ul care și-a terminat primul execuția este primul în coadă
    - task-ul care și-a terminat ultimul execuția este ultimul în coadă

Sistemul funcționează astfel:

Init

- Se definesc următoarele atribute (care vor fi citite din fișierul de intrare):
    - o cuantă de timp **Q** măsurată în milisecunde (ms) reprezentând timpul maxim de rulare continuă a unui thread
    - numărul de core-uri disponibile **C**, de unde deducem numărul maxim de thread-uri ce pot rula în paralel: **N = 2 * C**
- Se va crea pool-ul de thread-uri available în funcție de N.
- Se vor inițializa cozile de task-uri.

Rularea în paralel a task-urilor:

- Se preiau spre rulare task-uri din coada de așteptare până se golește coada de așteptare/nu mai există de thread-uri available în pool.
- Se rulează numai la comanda run pentru un timp dat ca parametru.
  Timpul poate fi: 1 <= Q <= T.
  Prin urmare, sistemul va rula thread-urile pentru maxim Q timp la un moment dat, după care va face verificările explicate mai jos. Practic, va fi un proces ciclic, cu pas Q.
- După expirarea unei cuante de timp / timpul de rulare rămas este mai mic decât cuanta :
    - pentru fiecare task care si-a terminat execuția, în acest timp:
        - ID-ul asociat task-ului devine disponibil și poate fi atribuit altor task-uri noi
        - acesta este mutat în coada de task-uri terminate
        - thread-ul asociat va fi eliberat și adăugat în pool
    - dacă au devenit thread-uri available și mai există task-uri în coada de așteptare, alocăm thread-uri și le inserăm în coada running până se golește coada de așteptare/nu mai există de thread-uri available în pool.
    - dacă nu mai există task-uri running și coada de așteptare e goală, se termină rularea curentă, chiar dacă timpul dat de comanda run nu s-a încheiat (nici nu se mai adaugă la timpul total pentru comanda finish).

4. Descrierea operațiilor și a datelor de intrare

Rezolvarea temei va consta în efectuarea unui set de operații descrise în fișierul de intrare.

A. Adăugare task în coada de așteptare:

Sintaxă:
`add_tasks <number_of_tasks> <exec_time_in_ms> <priority>`

Mod de funcționare: Creează un task nou cu atributele date și îl inserează în coada de așteptare după următoarele criteriile prezentate în secțiunea de 3.

În caz de succes afișează un mesaj de forma:
`Task created successfully : ID <task_id>.`

Exemplu:
`add_tasks 3 500 1`

```
Task created successfully : ID 1.
Task created successfully : ID 2.
Task created successfully : ID 3.
```

Se garantează că la orice moment de timp va fi disponibil cel puțin un ID pentru un task nou.

B. Determinarea stării unui task:

Sintaxă: `get_task <task_id>`

Exemplu: `get 123`

Mod de funcționare: Comanda caută toate task-urile cu identificatorul `<task_id>`. Prima dată se verifică dacă există task-ul în starea running, apoi task-urile din coada de așteptare, iar apoi task-urile din coada de task-uri terminate.
Datele se vor afișa în formatul de mai jos :

- Task în starea **running**:
  `Task <task_id> is running (remaining_time = <timp_ramas>).`
- Task în starea **waiting**:
  `Task <task_id> is waiting (remaining_time = <timp_ramas>).`
- Task în starea **finished**:
  `Task <task_id> is running (remaining_time = <timp_ramas>).`

În cazul în care nu a existat niciodată un task cu ID-ul dat :
`Task <task_id> not found.`

C. Determinarea stării unui thread:

Sintaxă: `get_thread <thread_id>`

Mod de funcționare: Pentru a determina starea thread-ului cu identificatorul `<thread_id>`, se verifică:

- dacă există thread-ul în pool-ul de threads: thread-ul este idle
- dacă rulează un task: se printează statusul task-ului

Datele se vor afișa în formatul de mai jos :

- Thread în starea **running**:
  `Thread <thread_id> is running task <task_id> (remaining_time = <timp_ramas>).`
- Thread găsit în pool:
  `Thread <thread_id> is idle.`

Exemplu: `get 2`

```
Thread 2 is running task 123 (remaining_time = 540).
```

Or

```
Thread 2 is idle.
```

D. Afișarea cozii de așteptare

Sintaxă: `print waiting`

Mod de funcționare: Afișează coada de așteptare a planificatorului la momentul curent în felul următor:

```
====== Waiting queue ======
[(<task_id1>: priority = <prioritate1>, remaining_time
=<timp_execuție_rămas1>),
...,
(<task_idN>: priority = <prioritateN>, remaining_time =
<timp_execuție_rămasN>)]
```

Exemplu:

```
====== Waiting queue ======
[(345: priority = 45, remaining_time = 500),
(4653: priority = 27, remaining_time = 100),
(1236: priority = 27, remaining_time = 220),
(2345: priority = 10, remaining_time = 165)]
```

E. Afișarea cozii running

Sintaxă: `print running`

Mod de funcționare: Afișează coada de așteptare a planificatorului la momentul curent în felul următor:

```
====== Running in parallel ======
[(<task_id1>: priority = <prioritate1>, remaining_time =
<timp_execuție_rămas1>, running_thread = <thread_id1>),
...,
(<task_idN>: priority = <prioritateN>, remaining_time =
<timp_execuție_rămasN>, running_thread = <thread_idN>)]
```

Exemplu:

```
====== Running in parallel ======
[(345: priority = 45, remaining_time = 500, running_thread = 2),
(4653: priority = 27, remaining_time = 100, running_thread = 1),
(1236: priority = 27, remaining_time = 220, running_thread = 5),
(2345: priority = 10, remaining_time = 165, running_thread = 6)]
```

F. Afișarea cozii de task-uri terminate

Sintaxă: `print finished`

Mod de funcționare: Afișează coada de așteptare a planificatorului la momentul curent în felul următor:

```
====== Finished queue ======
[(<task_id1>: priority = <prioritate1>, executed_time =
<timp_executat1>),
...,
(<task_idN>: priority = <prioritateN>, executed_time =
<timp_executatN>)]
```

Exemplu:

```
====== Finished queue ======
[(345: priority = 45, executed_time = 500),
(4653: priority = 27, executed_time = 100),
(1236: priority = 27, executed_time = 220),
(2345: priority = 10, executed_time = 165)]
```

G. Executarea unui număr de unități de timp

Sintaxă: `run <număr_unități_timp>`

Mod de funcționare: Execută următoarele T = `număr_unități_timp` pe procesor. Se va afișa:

`Running tasks for <număr_unități_timp> ms…`

H. Terminarea execuției tuturor task-urilor

Sintaxă: `finish`

Mod de funcționare: Se continuă execuția până când toate task-urile ajung în coada de task-uri terminate.
La finalul execuției, se afișează timpul total necesar pentru terminarea task-urilor neterminate.

în momentul apelării comenzii `finish`:
`Total time: <total_execution_time>`

În datele de intrare va exista maxim un apel al comenzii `finish`, iar după acesta nu vor mai exista alte comenzi.

5. Restricții și precizări:

- 1 <= Q <= 1000, dimensiunea unei cuante de timp este de maxim o secundă.
- 1 <= T <= 10 000 000, timpul dat ca parametru de comanda run.
- 1 <= C <= 64, max 64 cores.
- 0 <= linii în fișierul de intrare <= 10 000
- 0 <= număr total de task-uri <= 32767
- 0 <= priority <= 127
- se garantează că datele de intrare vor fi corecte
- nu aveți voie cu variabile globale
- programul va fi rulat astfel: `./tema2 in_file out_file`
- comenzile se citesc din fișierul **in_file**, iar rezultatele se scriu în fișierul **out_file**
- stivele și cozile vor fi implementate ca liste generice simplu sau dublu înlănțuite (eventual circulare)
- nu aveți voie sa iterați prin structuri; folosiți doar operațiile specifice pentru stive și cozi
- pentru sortări, afișări se vor folosi doar stive/cozi auxiliare (nu se permite iterarea prin stive/cozi)
- 40% din testele de intrare vor respecta următoarele condiții:
  - timpii de execuție ai proceselor vor fi multipli ai lui Q
  - argumentul comenzii run va fi multiplu al lui Q

6. Exemplu:

Intrare	Ieșire
500 // Q 1 // C => N = 2 add_tasks 2 1000 3 add_tasks 3 500 3 print waiting run 500 print finished print running run 250 print running get_task 1 get_thread 1 run 400 print running print finished finish	Task created successfully : ID 1. Task created successfully : ID 2. Task created successfully : ID 3. Task created successfully : ID 4. Task created successfully : ID 5. ====== Waiting queue ======: [(3: priority = 3, remaining_time = 500), (4: priority = 3, remaining_time = 500), (5: priority = 3, remaining_time = 500), (1: priority = 2, remaining_time = 1000), (2: priority = 2, remaining_time = 1000)] Running tasks for 500 ms… ====== Finished queue ====== [(3: priority = 3, executed_time = 500), (4: priority = 3, executed_time = 500)] ====== Running in parallel ====== [(5: priority = 3, remaining_time = 500, running_thread = 1), (1: priority = 2, remaining_time = 1000, running_thread = 0)] Running tasks for 250 ms… ====== Running in parallel ====== [(5: priority = 3, remaining_time = 250, running_thread = 1), (1: priority = 2, remaining_time = 750, running_thread = 0)] Task 1 is running (remaining_time = 750). Thread 1 is running task 5 (remaining_time = 250). Running tasks for 400 ms… ====== Running in parallel ====== [(1: priority = 2, remaining_time = 350, running_thread = 0), (2: priority = 2, remaining_time = 1000, running_thread = 1)] ====== Finished queue ====== [(3: priority = 3, executed_time = 500), (4: priority = 3, executed_time = 500), (5: priority = 3, executed_time = 500)] Total time: 2150

Explicație :

Avem numărul de cores C = 1 => Putem rula maxim N = 2 * C = 2 thread-uri în paralel. Stiva (pool-ul) de thread-uri va fi: 0 1, cu 0 în vârf.

print waiting:

Task-urile în coada de waiting sunt ordonate conform regulilor de la secțiunea 3.2.a).

run 500, print finished, print running :

Se preiau task-uri din waiting, coada de running fiind goală. Primul thread este thread-ul 0 si va prelua task-ul 3. Thread-ul 1 preia task-ul 4. În coada running, vor apărea în aceeași ordine. Se observă că T == Q, deci va fi nevoie de un singur ciclu. Se rulează pentru 500. După rulare, se verifică starea task-urilor și se observă ca ambele s-au terminat. Prin urmare, thread-ul 0 va fi pus în stivă primul, apoi thread-ul 1. În vârful stivei, va fi thread-ul 1. Pentru că acum avem thread-uri libere, preluăm următoarele task-uri din coada waiting: Thread-ul 1 preia task-ul 5, thread-ul 0 preia task-ul 1.

run 250:

Coada running rămâne neschimbată după acest run pentru că nu se termină niciun task. Se modifică doar timpii de execuție rămași task-urilor.

get_task 1:

Se caută task-ul după regulile de la secțiunea 4.B. Se găsește în coada running.

get_thread 1:

Se caută thread-ul după regulile de la secțiunea 4.C. Se găsește în coada running.

run 400, print running, print finished:

După 400 ms, se observă că task-ul 5 s-a încheiat. Thread-ul 1 este eliberat și adăugat în pool. Acum că avem un thread liber, adăugăm următorul task din waiting, rulând în cadrul thread-ului 1 (pop din stivă). Deoarece are un remaining time mai mare decât celalalt task deja în running, și are aceeași prioritate, acest task (2) va fi inserat ordonat, după task-ul 1.

În coada finished, au fost adăugate task-urile în ordinea în care s-au terminat.

finish:

Timpul de rulare de până acum a fost : 500 + 250 + 400 = 1150ms
În acest moment, avem în running două task-uri cu 350ms rămase, respectiv 1000ms.
Timpul necesar terminării acestor task-uri este echivalent cu run 1000 => 1000ms.
Timpul total va fi: 1150 + 1000 = 2150ms.

7. Notare:

- **135 de puncte** obținute pe testele de pe vmchecker. **Observatie**: Se pot obține punctaje parțiale fara a implementa toate comenzile (fiecare test valoreaza 5 puncte).
- **10 puncte:** coding style;
- **5 puncte:** README - va conține detaliile de implementare a temei, precum și **punctajul obținut la teste** (la rularea pe calculatorul propriu)
- **bonus: 20 de puncte** pentru soluțiile care nu au memory leak-uri (bonusul se acordă doar dacă testul a trecut cu succes);
- temele care nu compilează, nu rulează sau obțin punctaj 0 pe vmchecker, vor primi punctaj 0;
- se depunctează pentru:
    - warninguri la compilare (trebuie compilat cu -Wall);
    - linii mai lungi de 80 de caractere;
    - folosirea incorectă de pointeri;
    - lipsa verificărilor codurilor de eroare întoarse de funcții;
    - alte situații nespecificate aici, dar considerate inadecvate.

8. Reguli de trimitere a temelor

- temele vor trebui încărcate atât pe vmchecker (în secțiunea **Structuri de Date Seria CB)** cât și pe *curs.upb.ro***,** în secțiunea aferentă temei 3.
- arhiva cu rezolvarea temei trebuie să fie .zip și să conțină:
    - fișiere sursă (fiecare fișier sursă va trebui să inceapă cu un comentariu de forma: /* NUME Prenume - grupa */)
    - fișier **README,** denumit obligatoriu astfel, care să conțină detalii despre implementarea temei;
    - fișier **Makefile**, denumit obligatoriu astfel, cu două reguli:
        - **build**, care va compila sursele și va obține executabilul cu numele **tema2**
        - **clean**, care va șterge executabilele și alte fișier obiect generate
- arhiva trebuie să conțină doar fișierele sursă (inclusiv Makefile și README); nu se acceptă fișiere executabile sau obiect.
- dacă arhiva nu respectă specificațiile de mai sus nu va fi acceptată la upload și tema nu va fi luată în considerare.

9. Reguli împotriva copierii temelor

- se consideră copiate doua teme care seamănă suficient de mult pentru a putea trage aceasta concluzie;
- modificarea unei alte teme, asemănarea mai mult sau mai puțin evidentă a implementării, bucăți de cod identice etc. duc la considerarea temelor în cauză ca fiind copiate;
- pentru prima temă copiată: atât sursei cât și destinației li se va anula punctajul pentru tema respectivă şi ambii studenţi vor primi mustrare scrisă, fără discuții relative la cine a copiat de la cine și a cui e vina;
- la a doua temă copiată: atât sursei cât și destinației li se va anula punctajul pentru toate temele (ceea ce va duce la repetarea materiei) şi ambii studenţi vor primi mustrare scrisă, fără discuții relative la cine a copiat de la cine și a cui e vina.