**Universiteti i Prishtinës “Hasan Prishtina”**

**Fakulteti Inxhinierisë Elektrike dhe Kompjuterike**

****

**Dokumentim teknik i projektit**

**Lënda: Sistemet Operative**

**Titulli i projektit: Dining Philosophers problem**

**Emri profesorit/Asistentit Emri & mbiemri studentëve / email adresa**

|  |  |  |
| --- | --- | --- |
| Prof. Artan MAZREKAJ  As. Dalinë VRANOVCI | 1. Albina AHMET | [albina.ahmeti5@student.uni-pr.edu](mailto:albina.ahmeti5@student.uni-pr.edu) |
| 2. Alma LATIFI | [alma.latifi1@student.uni-pr.edu](mailto:alma.latifi1@student.uni-pr.edu) |
| 3. Festina GASHI | [festina.gashi6@student.uni-pr.edu](mailto:festina.gashi6@student.uni-pr.edu) |
| 4. Albiona VUKAJ | [albiona.vukaj@student.uni-pr.edu](mailto:albiona.vukaj@student.uni-pr.edu) |

Prishtinë, 2022

**Përmbajtja**

[Abstrakti 3](#_Toc94115197)

[1. Hyrje 4](#_Toc94115198)

[1.1. Bounded Buffer Problem 4](#_Toc94115199)

[1.2. Dining Philosophers Problem 4](#_Toc94115200)

[1.3. The Readers Writers Problem 4](#_Toc94115201)

[2. Qëllimi 5](#_Toc94115202)

[2.1. Vështrim i përgjithshëm 5](#_Toc94115203)

[2.2. Kufizimet dhe kushtet për problemin 5](#_Toc94115204)

[2.3. Zgjidhja 5](#_Toc94115205)

[2.3.1. Përpjekja e parë 6](#_Toc94115206)

[2.3.2. Përpjekja e dytë 6](#_Toc94115207)

[2.3.3. Përpjekja e tretë 7](#_Toc94115208)

[3. Pjesa kryesore 8](#_Toc94115209)

[3.1. Zbatimi në C++ 9](#_Toc94115210)

[3.2. Output 13](#_Toc94115211)

[4. Konkluzioni 14](#_Toc94115212)

[5. Referencat 15](#_Toc94115213)

Abstrakti

Në këtë projekt do të diskutojmë rreth njërit nga problemeve të ndryshme klasike të sinkronizimit. Semafori(Semaphore) mund të përdoret në probleme të tjera sinkronizimi përveç përjashtimit të ndërsjellë. Më poshtë janë disa nga problemet klasike që përshkruajnë të metat e sinkronizimit të procesit në sistemet ku proceset bashkëpunuese janë të pranishme.

Ne do të diskutojmë tre problemet e mëposhtme, mirëpo do të fokusohemi tek problemi i 2:

1. Bounded Buffer (Producer-Consumer) Problem
2. Dining Philosophers Problem
3. The Readers Writers Problem

Teknologjitë që kemi përdorur janë:

* Editori: Dev C++
* Gjuha programuese: C++
* Sistemi Operativ: Windows
* Vegla për përshkrimin e detyrës: MS Word

1. Hyrje

Fillimisht do të flasim për të tri problemet pastaj do ta shqyrtojmë problemin e dytë.

1. Bounded Buffer Problem

Për shkak se buffer pool ka një madhësi maksimale, ky problem shpesh quhet Bounded buffer problem.

* Ky problem përgjithësohet për sa i përket Producer Consumer problem, ku përdoret një grup buferik i fundëm për të shkëmbyer mesazhe ndërmjet proceseve të prodhuesit dhe konsumatorit.
* Zgjidhja e këtij problemi është krijimi i dy semaforëve numërues "plot" dhe "bosh" për të mbajtur gjurmët e numrit aktual të buferëve të plotë dhe të zbrazët respektivisht.
* Në këtë Prodhuesi kryesisht prodhon një produkt dhe konsumatorët e konsumojnë produktin, por të dy mund të përdorin çdo herë një nga kontejnerët.
* Kompleksiteti kryesor i këtij problemi është se ne duhet të mbajmë numërimin për kontejnerët bosh dhe të plotë që janë në dispozicion.

1. Dining Philosophers Problem

* Problemi i filozofit të ngrënies përfshin shpërndarjen e burimeve të kufizuara për një grup procesesh në një mënyrë pa ngërç dhe pa uria.
* Janë pesë filozofë të ulur rreth një tavoline, në të cilën ka pesë shkopinj/pirunë të mbajtur pranë tyre dhe një tas me oriz në qendër. Kur një filozof dëshiron të hajë, ai përdor dy shkopinj - një nga e majta dhe një nga e djathta. . Kur një filozof dëshiron të mendojë, ai i mban të dy shkopinjtë në vendin e tyre origjinal.

1. The Readers Writers Problem

* Në këtë problem ka disa procese (të quajtur lexues) që lexojnë vetëm të dhënat e përbashkëta, dhe nuk i ndryshojnë kurrë ato, dhe ka procese të tjera (të quajtur shkrimtarë) të cilët mund t'i ndryshojnë të dhënat përveç leximit, ose në vend që t'i lexojnë ato.
* Ekzistojnë lloje të ndryshme të problemeve lexues-shkrimtarë, më të përqendruar në prioritetet relative të lexuesve dhe shkrimtarëve.
* Kompleksiteti kryesor me këtë problem vjen nga lejimi i më shumë se një lexuesi për të hyrë në të dhënat në të njëjtën kohë

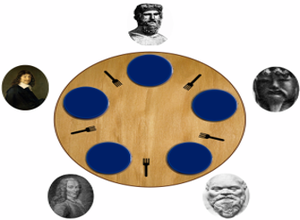
1. Qëllimi

Në shkencën kompjuterike, problemi i filozofëve të ngrënies është një problem që përdoret shpesh në hartimin e algoritmit për të ilustruar çështjet e sinkronizimit dhe teknikat për zgjidhjen e tyre. Fillimisht u formulua në vitin 1965 nga Edsger Dijkstra si një detyrë e provimit për studentët, i paraqitur në termat e kompjuterëve që konkurrojnë për akses në pajisjet periferike të kasetës(ose me bo paisje me shirit qysh doni). Menjëherë pas kësaj, Tony Hoare i dha problemit formulimin e tij aktual.

1. Vështrim i përgjithshëm

Problemi i Filozofëve të ngrënies: janë 5 filozofë të cilët janë të angazhuar në dy aktivitete të menduarit dhe të ngrënit. Shujtat merren në mënyrë të përbashkët në një tavolinë me pesë pjata dhe pesë pirunë në mënyrë ciklike siç tregohet në figurën 1.

1. Kufizimet dhe kushtet për problemin
2. Çdo filozof ka nevojë për dy pirunë për të ngrënë.
3. Çdo filozof mund të marrë pirunët në anë të majtë ose në anë të djathtë, por vetëm një pirun e jo të dyt në të njejtën kohë.
4. Filozofët hanë vetëm kur kanë dy pirunë. Ne duhet të hartojmë një protokoll të tillë, d.m.th. protokoll para dhe pas, i cili siguron që një filozof të hajë vetëm nëse ai ose ajo kishte dy pirunë.
5. Çdo pirun është ose i pastër ose i ndotur.



*Figura 1. Problemi i Filozofëve të ngrënies*

*Source:* [*Wikepedia*](https://en.wikipedia.org/wiki/Dining_philosophers_problem)

1. Zgjidhja

Karakteristikat e korrektësisë që duhet të plotësojëm për zgjidhjen e këtij problemi janë:

1. Parimi i përjashtimit të ndërsjellë - Nuk ka dy filozofë që mund t'i kenë dy pirunët në të njëjtën kohë.
2. Të lirë nga bllokimi - Çdo filozof mund të ketë mundësinë për të ngrënë në një kohë të caktuar të fundme.
3. Të lirë nga uria – Kur pak filozofë janë duke pritur, atëherë njeriu ka një shans për të ngrënë në një kohë.
4. Asnjë alternim i rreptë.
5. Shfrytëzimi i duhur i kohës.

**Algoritmi (skica):**

loop forever

p1: think

p2: preprotocol

p3: eat

p4: postprotoco

1. Përpjekja e parë

Supozojmë se çdo filozof është inicializuar me indeksin e tij dhe se shtimi është në mënyrë implicite modulo 5. Çdo pirun është modeluar si një semafor ku pritja korrespondon me marrjen e një piruni dhe sinjali korrespondon me vendosjen e një piruni.

**Algoritmi -**

semaphore array[0..4] fork ← [1, 1, 1, 1, 1]

loop forever

p1 : think

p2 : wait(fork[i])

p3 : wait(fork[i + 1])

p4 : eat

p5 : signal(fork[i])

p6 : signal(fork[i + 1])

**Problemi me këtë zgjidhje:**

Kjo zgjidhje mund të çojë në një bllokim nën një ndërthurje që i detyron të gjithë filozofët të marrin pirunët e tyre në anë të majtë përpara se ndonjëri prej tyre të përpiqet të marrë një pirun në anë të djathtë. Në këtë rast, të gjithë filozofët janë duke pritur për pirunin e duhur, por askush nuk do të ekzekutojë një udhëzim të vetëm.

1. Përpjekja e dytë

Një mënyrë për të trajtuar situatën e mësipërme është të kufizoni numrin e filozofëve që hyjnë në dhomë në katër. Duke e bërë këtë, njëri prej filozofëve përfundimisht do të marrë të dy pirunin dhe do të ekzekutojë të gjithë udhëzimet që çojnë në asnjë bllokim.

**Algoritmi -**

semaphore array[0..4] fork ← [1, 1, 1, 1, 1]

semaphore room ← 4

loop forever

p1 : think

p2 : wait(room)

p3 : wait(fork[i])

p4 : wait(fork[i + 1])

p5 : eat

p6 : signal(fork[i])

p7 : signal(fork[i + 1])

p8 : signal(room)

Në këtë zgjidhje, ne disi ndërhyjmë në problemin e dhënë pasi lejojmë vetëm katër filozofë.

1. Përpjekja e tretë

Ne përdorim algoritmin asimetrik në përpjekjen ku katër filozofët e parë ekzekutojnë zgjidhjen origjinale, por filozofi i pestë pret për pirunin në anë të djathtë dhe më pas për pirunin në anë të majtë.

**Algoritmi -**

semaphore array [0..4] fork ← [1,1,1,1,1]

**Për katër filozofët e parë-**

loop forever

p1 : think

p2 : wait(fork[i])

p3 : wait(fork[i + 1])

p4 : eat

p5 : signal(fork[i])

p6: signal(fork[i + 1])

**Për filozofin e pestë-**

loop forever

p1 : think

p2 : wait(fork[0])

p3 : wait(fork[4])

p4 : eat

p5 : signal(fork[0])

p6 : signal(fork[4])

Shënim - Kjo zgjidhje njihet edhe si **Zgjidhja Chandy/Mishra**.

1. Pjesa kryesore

Përparësitë e kësaj zgjidhjeje:

1. Lejon një shkallë të madhe të konkurrenës
2. Të lirë nga uria
3. Të lirë nga Bllokimi
4. Zgjidhje më fleksibël
5. Ekonomik
6. Drejtësia
7. Kufizueshmëria

Më sipër diskutoi zgjidhjen e problemit duke përdorur semaforin. Tani me monitorët, ku këtu, MONITOR ka kuptomin e një grupi pirunësh që numëron numrin e pirunëve falas të disponueshëm për çdo filozof. Operacioni take Forks pret në një variabël kushti derisa të jenë të disponueshme dy pirunë. Zvogëlon numrin e pirunëve në dispozicion të fqinjit të tij përpara se të largohet nga monitori. Pasi ka ngrënë, një filozof e quan versionin Forks, i cili përditëson pirunin e grupit dhe kontrollon nëse lirimi i këtyre pirunëve bën të mundur sinjalizimin.

**Algoritmi -**

monitor ForkMonitor:

integer array[0..4]

fork ← [2,2,2,2,2]

condition array[0..4]OKtoEat

operation takeForks(integer i)

if(fork[i]!=2)

waitC(OKtoEat[i])

fork[i+1]<- fork[i+1]-1

fork[i-1] <- fork[i-1]-1

operation releaseForks(integer i)

fork[i+1] <- fork[i+1]+1

fork[i-1] <- fork[i-1]

if(fork[i+1]==2)

signalC(OKtoEat[i+1])

if(fork[i-1]==2)

signalC(OKtoEat[i-1])

**Për çdo filozof -**

loop forever :

p1 : think

p2 : takeForks(i)

p3 : eat

p4 : releaseForks(i)

Këtu Monitor do të sigurojë të gjitha këto nevoja të përmendura më sipër.

1. Zbatimi në C++

Këtu është programi për të njëjtën duke përdorur monitorët në C++ si më poshtë:

// Header file include

#include <bits/stdc++.h>

#include <pthread.h>

#include <unistd.h>

using namespace std;

#define N 10

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (phnum + 4) % N

#define RIGHT (phnum + 1) % N

// Philosopher index

int phil[N];

int times = 200;

class monitor {

// state of the philosopher

int state[N];

// Philosopher condition variable

pthread\_cond\_t phcond[N];

// mutex variable for synchronization

pthread\_mutex\_t condLock;

public:

// Test for the desired condition

// i.e. Left and Right philosopher are not reading

void test(int phnum)

{

if (state[(phnum + 1) % 5] != EATING

and state[(phnum + 4) % 5] != EATING

and state[phnum] == HUNGRY) {

state[phnum] = EATING;

pthread\_cond\_signal(&phcond[phnum]);

}

}

// Take Fork function

void take\_fork(int phnum)

{

pthread\_mutex\_lock(&condLock);

// Indicates it is hungry

state[phnum] = HUNGRY;

// test for condition

test(phnum);

// If unable to eat.. wait for the signal

if (state[phnum] != EATING) {

pthread\_cond\_wait(&phcond[phnum], &condLock);

}

cout << "Philosopher " << phnum << " is Eating"

<< endl;

pthread\_mutex\_unlock(&condLock);

}

// Put Fork function

void put\_fork(int phnum)

{

pthread\_mutex\_lock(&condLock);

// Indicates that I am thinking

state[phnum] = THINKING;

test(RIGHT);

test(LEFT);

pthread\_mutex\_unlock(&condLock);

}

// constructor

monitor()

{

for (int i = 0; i < N; i++) {

state[i] = THINKING;

}

for (int i = 0; i < N; i++) {

pthread\_cond\_init(&phcond[i], NULL);

}

pthread\_mutex\_init(&condLock, NULL);

}

// destructor

~monitor()

{

for (int i = 0; i < N; i++) {

pthread\_cond\_destroy(&phcond[i]);

}

pthread\_mutex\_destroy(&condLock);

}

}

// Global Object of the monitor

phil\_object;

void\* philosopher(void\* arg)

{

int c = 0;

while (c < times) {

int i = \*(int\*)arg;

sleep(1);

phil\_object.take\_fork(i);

sleep(0.5);

phil\_object.put\_fork(i);

c++;

}

}

int main()

{

// Declaration...

pthread\_t thread\_id[N];

pthread\_attr\_t attr;

// Initialization...

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr,

PTHREAD\_CREATE\_JOINABLE);

for (int i = 0; i < N; i++) {

phil[i] = i;

}

// Creating...

for (int i = 0; i < N; i++) {

pthread\_create(&thread\_id[i], &attr, philosopher,

&phil[i]);

cout << "Philosopher " << i + 1 << " is thinking..."

<< endl;

}

// Joining....

for (int i = 0; i < N; i++) {

pthread\_join(thread\_id[i], NULL);

}

// Destroying

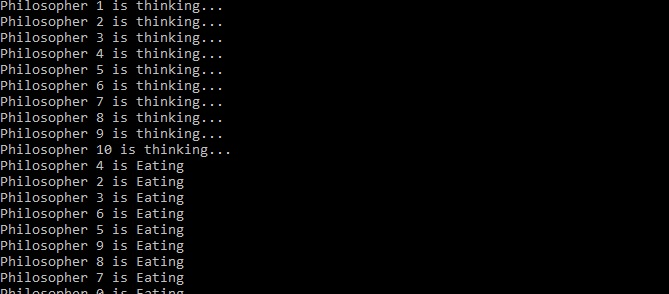
pthread\_attr\_destroy(&attr);

pthread\_exit(NULL);

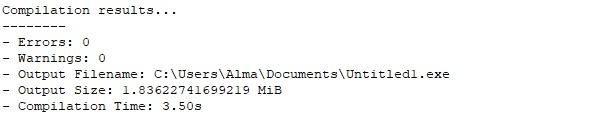
return 0;

}

1. Output



*Figura 2. Output for Dining Philosophers problem for the above program*



*Figura 3. Output for code*

**

*Video 1. Code and Output*

1. Konkluzioni

Zgjidh problemin e urisë. Etiketat e pastra/të pista veprojnë si një mënyrë për t'i dhënë përparësi proceseve më "të uritur" dhe një disavantazh për proceset që sapo kanë "ngrënë". Dikush mund të krahasojë zgjidhjen e tyre me atë ku filozofët nuk lejohen të hanë dy herë radhazi pa i lënë të tjerët të përdorin pirunët në mes. Zgjidhja e Chandy dhe Misra është më fleksibël se kaq, por ka një element që priret në atë drejtim.

Në analizën e tyre, ata nxjerrin një sistem të niveleve të preferencave nga shpërndarja e pirunëve dhe gjendjeve të tyre të pastra/të pista. Ata tregojnë se ky sistem mund të përshkruajë një graf jociklik të drejtuar, dhe nëse po, operacionet në protokollin e tyre nuk mund ta kthejnë atë grafik në një ciklik. Kjo garanton që bllokimi nuk mund të ndodhë. Megjithatë, nëse sistemi është inicializuar në një gjendje krejtësisht simetrike, si të gjithë filozofët që mbajnë pirunët e tyre në anën e majtë, atëherë grafiku është ciklik në fillim dhe zgjidhja e tyre nuk mund të parandalojë një bllokim. Inicializimi i sistemit në mënyrë që filozofët me ID më të ulët të kenë pirunë të ndotur siguron që grafiku të jetë fillimisht jociklik.

1. Referencat
2. <https://www.geeksforgeeks.org/implementing-race-condition-in-c/>
3. <https://www.sjsu.edu/people/robert.chun/courses/cs159/s3/Y.pdf>
4. <https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch05.html>
5. <https://www.studytonight.com/operating-system/classical-synchronization-problems>
6. <https://www.studytonight.com/operating-system/dining-philosophers-problem>
7. <https://www.geeksforgeeks.org/dining-philosophers-problem/>
8. <https://medium.com/swlh/the-dining-philosophers-problem-solution-in-c-90e2593f64e8>
9. <https://www.thecrazyprogrammer.com/2017/06/dining-philosophers-problem-c-c.html>
10. <https://codecharms.me/posts/operating-system-the-dining-philosophers-problem>
11. <https://cs.colby.edu/courses/F19/cs333/notes/9.ConcurrentProgramming(2).pdf>