

# Universiteti i Prishtinës “Hasan Prishtina”

## Fakulteti Inxhinierisë Elektrike dhe Kompjuterike



### Dokumentim teknik i projektit

Lënda: Sisteme Operative

Titulli i projektit: Dining Philosophers Problem

Emri profesorit/Asistentit

Emri & mbiemri studentëve / email adresa

Prof. Artan Mazrekaj Asistente. Mihrije Kadriu	1. Arbnora Dragaj	arbnora.dragaj@student.uni-pr.edu
	2. Rinesa Bejic	rinesa.bejic@student.uni-pr.edu
	3. Rinesa Gajtani	rinesa.gajtani@student.uni-pr.edu
	4. Shqiponje Dani	shqiponje.dani@student.uni-pr.edu
	5. Yllkë Berisha	yllke.berisha1@student.uni-pr.edu

Prishtinë, 2024

## Përmbajtja

I. Përshkrimi i problemit	3
I.I Problemi Dinning Philosophers	4
I.I.I Rrjedha e problemit	4
I.I.II Zgjidhja e problemit	5
II. Teknologjia	5
II. Implementimi	7
III.I Paraqitja e kodit me deadlock	7
III.I Implementimi i kodit (zgjidhja 1)	9
III.II Implementimi i kodit (zgjidhja 2)	17
Referencat	19

## I. Përshkrimi i problemit

Sinkronizimi i proceseve është procesi i koordinimit të veprimeve të shumë proceseve ose thread-ave për të siguruar që ato të punojnë së bashku në një mënyrë të përshtatshme. Një problem sinkronizimi ndodh kur shumë procese ose thread-a përpiqen të përdorin dhe të ndryshojnë burime të përbashkëta njëkohësisht, duke çuar në konflikte dhe veprime të paparashikuara. Metodat klasike të sinkronizimit përfshijnë përdorimin e semaforeve, mutex-ave dhe monitoreve. Ndër problemet klasike të sinkronizimit janë: Bounded-Buffer Problem, Readers-Writers Problem dhe Dining Philosophers Problem [1].

Semaforët janë mekanizma të sinkronizimit që përdoren për të kontrolluar qasjen në një burim të ndarë. Ato përdoren për të bërë që një proces të ndalojë deri sa një tjetër të përfundojë punën e tij me burimin e ndarë.

Mutex-at janë mekanizmat të sinkronizimit që përdoren për të kontrolluar qasjen eksklusive në një burim të ndarë. Ato garantojnë që vetëm një proces mund të ketë qasje në një burim në të njëjtën kohë.

Monitorët janë mekanizma të sinkronizimit që përdoren për të kontrolluar qasjen në një burim të ndarë dhe për të menaxhuar një grup procesesh që mund të jenë duke pritur për të pasur qasje në burimin e ndarë [1].

## I.I Problemi Dinning Philosophers

Problemi i darkës së filozofëve (Dinning Philosophers) është një problem që përdoret shpesh në hartimin e algoritmit për të ilustruar çështjen 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. Menjëherë pas kësaj Tony Hoare i dha problemit formulimin e tij aktual [1].

### I.I.I Rrjedha e problemit

Një problem klasik i konkurrencës ka të bëjë me një grup filozofësh të ulur në një tryezë të rrumbullakët duke ngrënë oriz. Numri shkopinjëve është i barabartë me numrin e filozofëve dhe një shkop është i vendosur në mes të çdo dy filozofëve. Domethënë secili filozof ka një shkop në të majtë dhe një në të djathtë. Ky problem ka të bëjë me paraqitjen e **think-eat** loop. Pasi filozofi ulet i qetë duke menduar për një kohë, një filozof merr uri. Për të ngrënë, një filozof kap shkopin në të majtë, kap shkopin në të djathtë, ha për një kohë duke përdorur të dy shkopinjtë, dhe më pas kthen shkopinjtë në tryezë dhe kthehet në të menduarit. Ekziston një gjendje e **deadlock** nëse filozofëve iu lejohej të kapin shkopinjtë lirshëm. Deadlock paraqitet pasi askush nuk është duke ngrënë, dhe pas një momenti të gjithë filozofët kapin shkopinjtë e tyre që gjenden në anën e majtë, dhe pastaj presin për shkopin e djathtë. Në këtë analogji paraqitet deadlock [1].

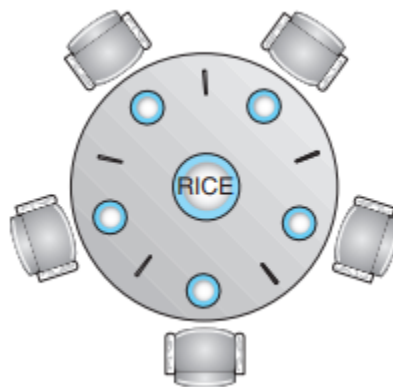


Figura 1: Situata në Dining Philosophers problem

### I.I.II Zgjidhja e problemit

Ekzistojnë disa mënyra ose metoda se si të zgjedhim problemin "Dining Philosophers Problem" dhe të parandalojmë deadlock-un.

Disa nga më të zakonshmet përfshijnë:

- Resource Hierarchy: kjo zgjidhje cakton një rend hierarkik për burimet (shkopinjët) të tillë që çdo filozof mund të marrë shkopin në të djathtë vetëm nëse e ka shkopin në të majtë. Kjo siguron që të ketë gjithmonë një shkop në dispozicion për çdo filozof, duke parandaluar deadlock-un.
- Chandy-Misra: Kjo zgjidhje i lejon filozofët të kërkojnë leje për të marrë shkopin dhe kërkesa jepet nga menaxheri i burimeve nëse dhe vetëm nëse mund të jepet pa shkaktuar deadlock.
- Time-Based: Kjo zgjidhje lejon çdo filozof të marrë shkopinjtë e tij për një periudhë të caktuar kohore, pas së cilës duhet t'i lëshojnë ato. Kjo siguron që asnjë filozof nuk mund të mbajë një shkop pafundësisht, duke parandaluar deadlock.
- Deadlock-Free: Kjo zgjidhje siguron që filozofët të mund të marrin shkopinjtë e tyre dhe të hanë në një mënyrë që deadlock-u të mos ndodhë duke i lejuar filozofët të marrin shkopinjtë në një rend rrethor.

Të gjitha zgjidhjet e mësipërme janë mënyra të ndryshme për të zgjidhur problemin e Dining Philosophers dhe për të parandaluar bllokimin, por ato kanë avantazhet dhe disavantazhet e tyre. Është e rëndësishme për të kuptuar plotësisht problemin dhe për të zgjedhur zgjidhjen e duhur për kërkesat specifike të problemit [2].

### II. Teknologjia

Për demonstrimin e projektit kemi përdorur gjuhën programuese C.

Gjuha C është e përshtatshme për këtë projekt sepse ajo mund të përdoret për të implementuar një sistem të saktë të mutex (mutual exclusion) dhe të kontrollit të kohës së pritjes, të cilat janë thelbësore për të zgjidhur problemin "Dining Philosophers Problem". Gjuha C gjithashtu mund të përdoret për të implementuar një logjikë të qartë dhe të thjeshtë të kontrollit të rradhës së filozofëve.

Nuk është përdorur ndonjë editor i veçantë, i tërë projekti është shkruar në një fajll të thjeshtë në Ubuntu. Ubuntu është një derivim i Linux-it me burim të hapur. Për ekzekutimin e kodit të shkruar e kemi instaluar në sistemin tonë një C compailer (përpilues) siç është GCC (GNU Compiler Collection) [2].

## II. Implementimi

### III.I Paraqitja e kodit me deadlock

Ky është një shembull i problemit "Dining Philosophers" në situatën me deadlock, sepse i mungon kushti i testimit nëse shkopi tjetër është i disponueshëm përpara se ta marrë atë dhe do të shkaktojë një deadlock sepse çdo filozof do të presë për shkopin që mbahet nga filozofi i ulur pranë tyre. Poshtë është paraqitur problemi i deadlock-ut me anë të kodit në C [1].

Rezultatet e arritura pas ekzekutimit të kodit:

```
yllka@yllka-VirtualBox:~/Desktop/projekti$ ./dining_deadlock
Philosopher 0 is thinking.
Philosopher 0 is eating.
Philosopher 1 is thinking.
Philosopher 2 is thinking.
Philosopher 2 is eating.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 0 is thinking.
Philosopher 0 is eating.
Philosopher 2 is thinking.
Philosopher 2 is eating.
Philosopher 0 is thinking.
Philosopher 0 is eating.
Philosopher 2 is thinking.
Philosopher 2 is eating.
Philosopher 0 is thinking.
Philosopher 2 is thinking.
Philosopher 2 is eating.
Philosopher 2 is thinking.
Philosopher 2 is eating.
Philosopher 2 is thinking.
Philosopher 2 is eating.
Philosopher 2 is thinking.
Philosopher 2 is eating.
Philosopher 2 is thinking.
Philosopher 2 is eating.
Philosopher 2 is thinking.
Philosopher 2 is eating.
```

Figura 2: Rezultatet e arritura nga kodi me deadlock

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <pthread.h>
4
5  #define NUM_PHILOSOPHERS 5
6
7  pthread_mutex_t chopsticks[NUM_PHILOSOPHERS];
8
9  void *philosopher(void *num) {
10     int id = *((int *)num);
11     while (1) {
12         printf("Philosopher %d is thinking.\n", id);
13         pthread_mutex_lock(&chopsticks[id]);
14         pthread_mutex_lock(&chopsticks[(id + 1) % NUM_PHILOSOPHERS]);
15         printf("Philosopher %d is eating.\n", id);
16         sleep(3);
17         pthread_mutex_unlock(&chopsticks[(id + 1) % NUM_PHILOSOPHERS]);
18         pthread_mutex_unlock(&chopsticks[id]);
19     }
20 }
21
22 int main() {
23     int i;
24     int nums[NUM_PHILOSOPHERS];
25     pthread_t threads[NUM_PHILOSOPHERS];
26
27     // Initialize mutexes
28     for (i = 0; i < NUM_PHILOSOPHERS; i++) {
29         pthread_mutex_init(&chopsticks[i], NULL);
30     }
31
32     // Create philosopher threads
33     for (i = 0; i < NUM_PHILOSOPHERS; i++) {
34         nums[i] = i;
35         pthread_create(&threads[i], NULL, philosopher, &nums[i]);
36     }
37
38     // Join philosopher threads
39     for (i = 0; i < NUM_PHILOSOPHERS; i++) {
40         pthread_join(threads[i], NULL);
41     }
42
43     return 0;
44 }
45
```

Figura 3: Kodi me deadlock



### III.I Implementimi i kodit (zgjidhja 1)

#### Resource hierarchy

Zgjidhja permes resource hierarchy ,ky program përdor semaforë dhe threda për të simuluar problemin Dining Philosophers në një mënyrë të sigurt dhe efikase. Ai e zgjidh problemin duke siguruar që filozoft mund të hajë në të njëjtën kohë dhe duke përdorur semaforë për të kontrolluar aksesin në gjendjen e përbashkët të problemit.

Fillimi i programit C, që përfshin librarite e nevojshme.

```
1 #include <pthread.h>
2 #include <semaphore.h>
3 #include <stdio.h>
4 #include <unistd.h>
5
```

**#include <pthread.h>** është një librari që permban deklaratat POSIX thread, e cila ofron funksione për krijimin, përfundimin dhe sinkronizimin e thredave.

**#include <stdio.h>** është një librari që qëndron për hyrje/dalje standarde dhe ofron funksione për futjen dhe nxjerrjen e të dhënave.

**#include <unistd.h>** akses në funksione të ndryshme të sistemit operativ , ne rastin tone sleep

**#include <stdlib.h>** libraria përdoret për funksione të ndryshme si ato për konvertimin e të dhënave

```
6 #define N 5 // Number of philosophers
7 #define THINKING 2
8 #define HUNGRY 1
9 #define EATING 0
10 #define LEFT (philosopher_num + N - 1) % N
11 #define RIGHT (philosopher_num + 1) % N
12
13 int state[N];
14 int philosopher_num[N] = {0, 1, 2, 3, 4};
15
16 sem_t mutex;
17 sem_t S[N];
```

**#define N 5** është një preprocessor directive që i jep vlerën N 5. Kjo vlerë është përdorur për të përcaktuar numrin e filozofëve në problem.

**#define THINKING 2, #define HUNGRY 1, #define EATING 0** janë preprocessor directives që i japin vlerave 2, 1 dhe 0 variabla THINKING, HUNGRY, EATING. Këto janë përdorur për të përshkruar gjendjen e filozofëve në problem.

**#define LEFT (philosopher\_id + N-1) % N, #define RIGHT (philosopher\_id + 1) % N** janë preprocessor directives që i japin vlera right dhe left.

(%)përdoret për të siguruar që vlera e LEFT dhe RIGHT të jetë gjithmonë brenda intervalit nga 0 në N-1. Qëllimi i kësaj është të sigurojë që filozofi i ulur në të majtë të filozofit aktual të jetë gjithmonë brenda intervalit të numrit të filozofëve në problem.

**int state[N]** është një array që ruan gjendjen e filozofëve, RIGHT përdoret për të gjetur filozofin të ulur në të djathtë të filozofit aktual.

**int philosophers[N] = { 0, 1, 2, 3, 4 }** është një array që ruan numrat e filozofëve.

**sem\_t mutex** është një semafor që përdoret për të kontrolluar aksesin në seksionin kritik të kodit.

**sem\_t S[N]** është një array e semaforëve që përdoren për të kontrolluar aksesin në shkopinjte e filozofëve të caktuar

```
88
89 int main()
90 {
91     int i;
92     pthread_t thread_id[N];
93
94     sem_init(&mutex, 0, 1);
95
```

Funksioni kryesor ku fillon ekzekutimi i programit

**int i** një variabël i të tipit integer, i cili përdoret si numërues cikli në program.

**pthread\_t thread\_id[N]** është një array që ruan identifikuesit e thread-ve të filozofëve. Këto identifikues janë të nevojshëm për të kontrolluar dhe menaxhuar thread-et në aplikacion.

**sem\_init(&mutex, 0, 1)** është një funksion që inicializon semaforin mutex me vlerën 1. Kjo do të thotë që semafori fillimisht është i lirë për të pranuar një request.

```
96     for (i = 0; i < N; i++)
97         sem_init(&S[i], 0, 0);
98
```

Ky cikël përsëritet nëpër të gjithë elementët e grupit S dhe inicializon çdo element të grupit si një semafor me vlerë 0 duke përdorur funksionin `sem_init(&S[i], 0, 0)`. Do të thotë që fillimisht të gjithë semaforët janë të kyçur dhe asnjë filozof nuk mund të marrë shkopinj. Kjo përdoret për të siguruar që filozofët mund të marrin shkopinj vetëm kur u lejohet ta bëjnë këtë, dhe kjo ndihmon në parandalimin e deadlock.

```
98
99     for (i = 0; i < N; i++) {
100 |
101         pthread_create(&thread_id[i], NULL,
102                        philosopher, &philosopher_num[i]);
103         printf("Philosopher %d is thinking\n", i + 1);
104     }
105
```

Ky cikël iteron nëpër të gjithë elementet e array-it `philosopher_num`. Për secilën iterim, krijon një thread të ri duke përdorur funksionin `pthread_create(&thread_id[i], NULL, philosopher, &philosopher_num[i])`.

Argumenti i parë `&thread_id[i]` është një pointer për identifikuesin e thread-it, argumenti i dytë `NULL` përdoret për të caktuar atributet e thread-it (atributet default i merr), argumenti i tretë `philosopher` është funksioni që thread-i do të ekzekutojë dhe argumenti i katërt `&philosopher_num[i]` është një pointer për argumentin e tipit integer që funksioni `philosopher` merr. Kjo është adresa e elementit `i`-të të vargut `philosopher_num` dhe përdoret për me identifikim threadin.

Pra, cikli for po krijon N thread, çdo thread përfaqëson një filozof, dhe funksioni që kryen thread është funksioni filozof.

```
105
106     for (i = 0; i < N; i++)
107         pthread_join(thread_id[i], NULL);
108 }
109
```

Ky kod pret që të gjithë thread-et e filozofëve të përfundojnë ekzekutimin e tyre.

Loopi for kalon nga 0 deri në N dhe në secilin iteracion, thirrret funksioni **pthread\_join(threads\_id[i], NULL)**.

**pthread\_join** është një funksion që pret për përfundimin e një thread-i të caktuar. Ajo merr dy argumente:

**threads\_id[i]**: ID-ja e thread-it që pret të përfundojë.

**NULL**: një pointer në një vend ku do të ruhet gjendja e daljes së thread-it kur përfundon ekzekutimin e tij. Ky argument përdoret për të marrë vlerën që thread-i ktheu kur përfundoi ekzekutimin e tij. Në këtë rast, është vendosur si NULL, duke thënë se gjendja e daljes së thread-it nuk ruhet.

Kjo është e rëndësishme sepse kjo siguron që programi i tërë nuk përfundon derisa të gjithë filozofët të përfundojnë së ngrëni dhe menduari, duke parandaluar ndërprerjen e programit para se filozofët të përfundojnë punën e tyre.

```
18
19 void test(int philosopher_num)
20 {
21     if (state[philosopher_num] == HUNGRY
22         && state[LEFT] != EATING
23         && state[RIGHT] != EATING) {
24         // state that eating
25         state[philosopher_num] = EATING;
26
27         sleep(1);
28
29         printf("Philosopher %d takes fork %d and %d\n",
30              philosopher_num + 1, LEFT + 1, philosopher_num + 1);
31
32         printf("Philosopher %d is Eating\n", philosopher_num + 1);
33
34
35         sem_post(&S[philosopher_num]);
36     }
37 }
38
```

Funksioni **void test(int philosopher\_num)** është një funksion ndihmës që përdoret për të testuar nëse një filozof mund ta merr shkopin dhe të fillojë të hajë. Ky funksion merr një argument, **philosopher\_num**, i cili është indeksi i filozofit në array.

Funksioni fillon duke kontrolluar nëse filozofi i tanishëm është i uritur dhe se filozofët që ulen në të majtë dhe në të djathtë nuk po hajjnë. Nëse kjo është e vërtetë, funksioni pastaj ndryshon gjendjen e filozofit në EATING, sleep për 1 sekond, dhe tregon në konsolë se filozofi ka marrë shkopinjet dhe ka filluar të hajë dhe pastaj liron semaforin **sem\_post(&S[philosopher\_num])**.

Funksioni **sem\_post()** rrit vlerën e semaforit. Nëse vlera e semaforit është aktualisht zero, atëherë një nga thread-et që është bllokuar në atë semafor është zhbllokuar.

Në këtë rast, semafori përdoret për të siguruar që vetëm një filozof mund të hajë në të njëjtën kohë. Kur një filozof mbaron së ngrëni, semafori lëshohet. Kjo lejon një filozof tjetër të marrë shkopinjtë dhe të fillojë të hajë.

```
38
39 void take_fork(int philosopher_num)
40 {
41     sem_wait(&mutex);
42
43     state[philosopher_num] = HUNGRY;
44
45     printf("Philosopher %d is Hungry\n", philosopher_num + 1);
46
47     test(philosopher_num);
48
49     sem_post(&mutex);
50
51     sem_wait(&S[philosopher_num]);
52
53     sleep(1);
54 }
55
```

Funksioni **void take\_fork(int philosopher\_id)** përdoret për të simuluar procesin e një filozofi që përpiqet të marrë shkopinjtë dhe të fillojë të hajë. Duhet një argument, **philosopher\_num**, i cili është indeksi i filozofit.

Funksioni fillon duke përdorur funksionin **sem\_wait()** për të marrë mutex-in e semaforit. Semafori mutex përdoret për të siguruar që vetëm një filozof mund të aksesojë gjendjen e përbashkët të problemit në të njëjtën kohë.

Hapi tjetër është ndryshimi i gjendjes së filozofit aktual në HUNGRY.

Funksioni më pas thërret funksionin **test()** për të kontrolluar nëse filozofi është në gjendje të marrë shkopinj dhe të fillojë të hajë. Nëse filozofi është në gjendje të fillojë të hajë, funksioni **test()** do të ndryshojë gjendjen e filzofit në EATING , do të simulojë kohën e ngrënies dhe do të printojë një mesazh.

Pas funksionit **test()**, funksioni përdor funksionin **sem\_post()** për të lëshuar semaforin mutex. Kjo u lejon filozofëve të tjerë të aksesojnë gjendjen e përbashkët dhe të marrin shkopinj nëse janë në gjendjen HUNGRY.

Më pas, funksioni përdor funksionin **sem\_wait()** për të marrë semaforin **S[philosopher\_id]**.

Funksioni **sem\_wait()** zvogëlon vlerën e semaforit. Nëse vlera e semaforit është aktualisht zero, atëherë thread-i që thërret funksionin do të bllokohet derisa semafori të lëshohet nga një thread tjetër që thërret funksionin **sem\_post()**.

Në këtë rast, semafori **S[philosopher\_id]** përdoret për të siguruar që vetëm një filozof mund të hajë në të njëjtën kohë. Kur një filozof përpiket të marrë shkopinj, funksioni do të thërrasë **sem\_wait(&S[philosopher\_id])** për të marrë semaforin. Nëse semafori tashmë mbahet nga një filozof tjetër, thredi do të bllokohet derisa semafori të lëshohet.

```
56
57 void put_fork(int philosopher_num)
58 {
59     sem_wait(&mutex);
60
61     state[philosopher_num] = THINKING;
62
63     printf("Philosopher %d putting fork %d and %d down\n",
64           philosopher_num + 1, LEFT + 1, philosopher_num + 1);
65     printf("Philosopher %d is thinking\n", philosopher_num + 1);
66
67     test(LEFT);
68     test(RIGHT);
69
70     sem_post(&mutex);
71 }
72
```

Funksioni **void put\_fork(int philosopher\_id)** përdoret për të simuluar procesin e një filozofi që hedh poshtë shkopinj pasi të ketë mbaruar së ngrëni. Duhet një argument, **philosopher\_num**, i cili është indeksi i filzofit .

Funksioni fillon duke përdorur funksionin **sem\_wait()** për të marrë mutex-in e semaforit.

Hapi tjetër është ndryshimi i gjendjes së filozofit aktual në THINKING duke përditësuar grupin e gjendjes. Kjo tregon se filozofi ka mbaruar së ngrëni dhe tani po mendon.

Më pas, funksioni thërret funksionin test() dy herë, një herë për filozofin e ulur në të majtë dhe një herë për filozofin që ulet në të djathtë. Kjo bëhet për të kontrolluar nëse njëri prej filozofëve që ulet në të majtë ose në të djathtë të filozofit aktual mund të fillojë të hajë tani që filozofi aktual ka hedhur poshtë shkopinj.

Pas funksionit test(), funksioni përdor funksionin **sem\_post()** për të lëshuar semaforin mutex. Kjo u lejon filozofëve të tjerë të aksesojnë gjendjen e përbashkët dhe të vendosin ose të marrin shkopinj nëse janë në gjendjen THINKING ose HUNGRY.

```
72
73 void* philosopher(void* num)
74 {
75     while (1) {
76
77         int* i = num;
78
79         sleep(1);
80
81         take_fork(*i);
82
83         sleep(0);
84
85         put_fork(*i);
86     }
87 }
88
```

Funksioni **void\* philosopher(void\* num)** është funksioni kryesor që simulon sjelljen e një filozofi. Duhet një pointer num që tregon indeksin e filozofit në grupin e filozofëve.

Funksioni fillon duke hyrë në një loop të pafund që simulon sjelljen e vazhdueshme të një filozofi. Brenda ciklit, funksioni fillimisht përdor një pointer **i** për të kthyer pointerin **void num** në një numër të plotë, që përfaqëson indeksin e filozofit.

Më pas, funksioni përdor funksionin **sleep(1)** për të simuluar kohën që i duhet filozofit për të filluar të mendojë.

Pastaj funksioni thërret funksionin **take\_fork()** për të simuluar procesin e një filozofi që përpiqet të marrë shkopinj dhe të fillojë të hajë.

Më pas, funksioni përdor sërish funksionin **sleep()**, kjo është bërë për të simuluar kohën që i duhet filozofit për të mbaruar së ngrëni.

Më në fund, funksioni thërret funksionin **put\_fork()** për të simuluar procesin e një filozofi që vendos shkopinj pasi të ketë mbaruar së ngrëni.

Rezultatet e ekzekutimit të kodit:

```
yllka@yllka-VirtualBox:~/Desktop/projekti$ ./dining_sem
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is Hungry
Philosopher 2 is Hungry
Philosopher 5 is Hungry
Philosopher 3 is Hungry
Philosopher 4 is Hungry
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 4 is Hungry
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 3 is Hungry
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 takes fork 5 and 1
```

Figura: Rezultatet e arritura nga kodi me semafor

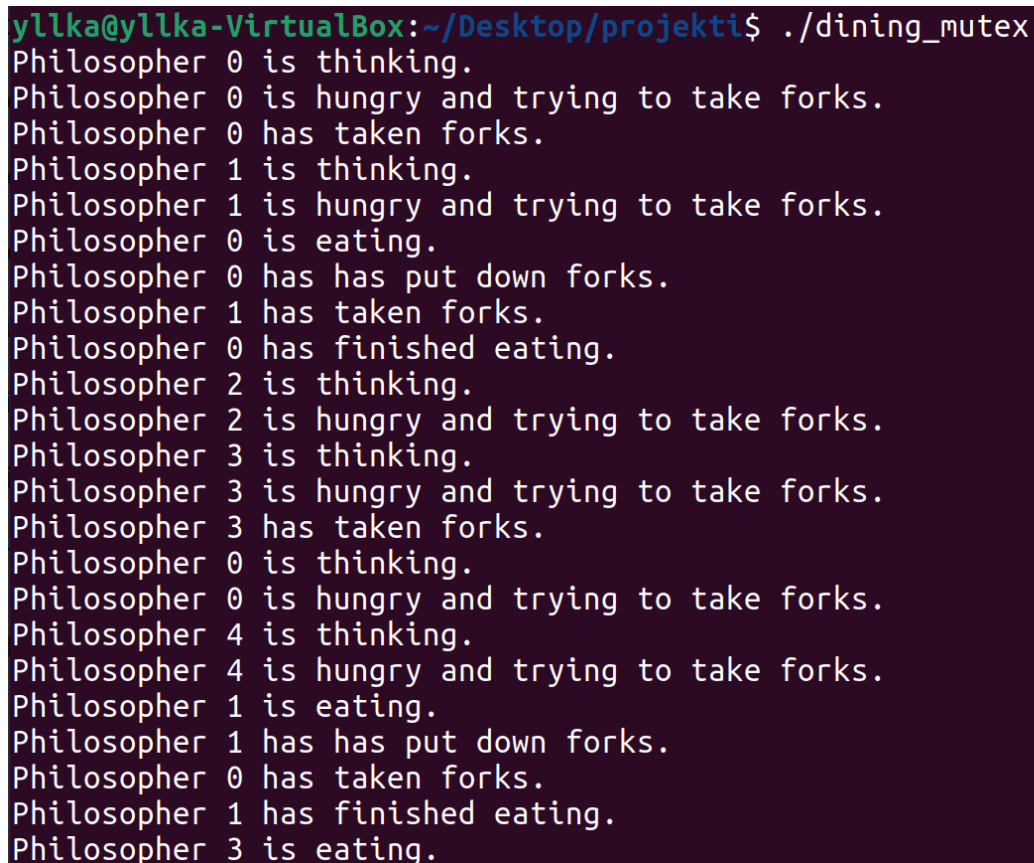


### III.II Implementimi i kodit (zgjidhja 2)

Ky kod për zgjidhjen e problemit të filozofëve që hanë përdor mutex dhe kushtet e variablave (condition variables) për të kontrolluar veprimet e filozofëve dhe përdorimin e pirunëve. Çdo filozof përfaqësohet nga një fije (thread) dhe ka tri gjendje të mundshme: duke menduar, i uritur dhe duke ngrënë. Kur një filozof është i uritur, ai përpiqet të marrë pirunët e tij të majtë dhe të djathtë. Nëse pirunët fqinj nuk janë në përdorim (fqinjët nuk janë duke ngrënë), ai mund të hajë. Përdoren mutex-at për të siguruar që ndryshimet në veprimet e filozofëve dhe përdorimin e pirunëve, dhe condition variables për të sinjalizuar kur një filozof mund të hajë.

Dallimi kryesor midis kësaj metode dhe metodës së parë është se kjo zgjidhje përdor condition variables të lidhura me secilin filozof për të sinjalizuar kur një filozof mund të hajë, ndërsa metoda e parë përdor semaforët për të arritur të njëjtin qëllim. Përdorimi i condition variables lejon një kontroll më të sofistikuar dhe të hollësishëm të veprimeve të filozofëve dhe bashkëveprimin e tyre, ndërsa përdorimi i semaforëve është një metodë më e drejtpërdrejtë dhe e thjeshtë për menaxhimin e këtyre veprimeve.

Rezultatet e ekzekutimit të kodit:



```
yllka@yllka-VirtualBox:~/Desktop/projekti$ ./dining_mutex
Philosopher 0 is thinking.
Philosopher 0 is hungry and trying to take forks.
Philosopher 0 has taken forks.
Philosopher 1 is thinking.
Philosopher 1 is hungry and trying to take forks.
Philosopher 0 is eating.
Philosopher 0 has has put down forks.
Philosopher 1 has taken forks.
Philosopher 0 has finished eating.
Philosopher 2 is thinking.
Philosopher 2 is hungry and trying to take forks.
Philosopher 3 is thinking.
Philosopher 3 is hungry and trying to take forks.
Philosopher 3 has taken forks.
Philosopher 0 is thinking.
Philosopher 0 is hungry and trying to take forks.
Philosopher 4 is thinking.
Philosopher 4 is hungry and trying to take forks.
Philosopher 1 is eating.
Philosopher 1 has has put down forks.
Philosopher 0 has taken forks.
Philosopher 1 has finished eating.
Philosopher 3 is eating.
```

Figura: Rezultatet e arritura nga kodi me mutex

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5
6 #define N 5 // Number of philosophers
7
8 // Possible states of philosophers
9 #define THINKING 0
10 #define HUNGRY 1
11 #define EATING 2
12
13 pthread_mutex_t forks[N];
14 pthread_cond_t conditions[N];
15 int states[N]; // States of philosophers
16
17 void *philosopher(void *num);
18 void take_forks(int phil);
19 void put_forks(int phil);
20 void test(int phil);
21
22 int main() {
23     pthread_t thread_id[N];
24     int i;
25
26     for (i = 0; i < N; i++) {
27         pthread_mutex_init(&forks[i], NULL);
28         pthread_cond_init(&conditions[i], NULL);
29         states[i] = THINKING; // Initial state: thinking
30     }
31
32     for (i = 0; i < N; i++) {
33         pthread_create(&thread_id[i], NULL, philosopher, &i);
34         sleep(1); // to avoid race conditions in printing
35     }
36
37     for (i = 0; i < N; i++) {
38         pthread_join(thread_id[i], NULL);
39     }
40
41     for (i = 0; i < N; i++) {
42         pthread_mutex_destroy(&forks[i]);
43         pthread_cond_destroy(&conditions[i]);
44     }
45
46     return 0;
47 }
48
49 void *philosopher(void *num) {
50     int phil = *(int *)num;
51
52     while (1) {
53         sleep(2); // Thinking for 2 seconds
54         printf("Philosopher %d is thinking.\n", phil);
55         take_forks(phil); // Take forks to eat
56         sleep(2); // Eating for 2 seconds
57         printf("Philosopher %d is eating.\n", phil);
58         put_forks(phil); // Put forks after eating
59         printf("Philosopher %d has finished eating.\n", phil);
60     }
61 }
62
63 void take_forks(int phil) {
64     pthread_mutex_lock(&forks[phil]);
65     states[phil] = HUNGRY; // Change state to hungry
66     printf("Philosopher %d is hungry and trying to take forks.\n", phil);
67     test(phil); // Test if philosopher can eat
68     while (states[phil] != EATING) { // While not eating
69         pthread_cond_wait(&conditions[phil], &forks[phil]); // Wait for signal
70     }
71     pthread_mutex_unlock(&forks[phil]);
72 }
73
74 void put_forks(int phil) {
75     pthread_mutex_lock(&forks[phil]);
76     states[phil] = THINKING; // Change state to thinking
77     printf("Philosopher %d has put down forks.\n", phil);
78     test((phil + N - 1) % N); // Test left neighbor
79     test((phil + 1) % N); // Test right neighbor
80     pthread_mutex_unlock(&forks[phil]);
81 }
82
83 void test(int phil) {
84     if (states[phil] == HUNGRY && states[(phil + N - 1) % N] != EATING && states[(phil + 1) % N] != EATING) {
85         states[phil] = EATING; // Change state to eating
86         printf("Philosopher %d has taken forks.\n", phil);
87         pthread_cond_signal(&conditions[phil]); // Signal that this philosopher can eat
88     }
89 }
90
```

## Referencat

---

- [1] Abraham Silberchatz, Peter Baer Galvin, Greg Gagne, Operating System Concepts 10th edition, Ęiley, 2018.
- [2] Ilya Sutskever, Greg Brockman, Ęojciech Zaremba and John Schulman, November 2022. [Online]. Available: <https://chat.com/chat>. [Accessed 16 January 2023].