1. K1(2019) Написати и објасните test and test and set решење за критичну секцију. Уколико би уместо TS(var) постојала операција SWAP која би недељиво обављала замену вредности два операнда (SWAP(var1, var2) : < temp = var1; var1 = var2; var2 = temp; >), да ли је могуће направити Fine grain решење и ако је могуће – направите га.

//COURSE GRAINED

bool lock = false;

process CS[i = 0 to n] {

<await (!lock) lock = true>

// critical section

lock = false;

}

//FINE GRAINED

bool lock = false;

process CS[i = 0 to n] {

do {

while(lock) skip;

} while(TS(lock));

// critical section

lock = false;

}

Ideja za Test and Test and Set je da prvo testiramo sa obicnim pristupom promenjivoj umesto da koristimo TS,

zato sto TS izaziva zagusenje kesa, zbog toga sto prilikom izvrsavanja potrebno je invalidovati/updejtovati

vrednost promenjive u kesevima svih jezgara.

//FINE GRAINED

bool lock = false;

bool TS(lock) {

bool var = true;

SWAP(lock, var);

return var;

}

process CS[i = 0 to n] {

do {

while(lock) skip;

} while(TS(lock));

// critical section

lock = false;

}

2. K1(2019)

Посматра се проблем синхронизације на баријери N процеса, који у бесконачној петљи извршавају неки посао. Након завршетка посла треба синхронизовати те процесе, тј. ниједан процес не сме да настави у нову итерацију извршавања посла ако сви процеси нису завршили претходну итерацију. Уколико не постоји процес координатора, написати код којим се решава дати проблем, користећи тачно N семафора за синхронизацију. Прокоментарисати особине приложеног решења.

// Najbolje resenje

sem waiting[N] = {1,0,0,...,0};

void process(int id) {

while (true) {

work();

wait(waiting[id]);

signal(waiting[(id+1)%N]);

wait(waiting[id]);

signal(waiting[(id+1)%N]);

}

}

// Drugi nacin

void process(int id) {

while (true) {

work();

if (id ==0) {

wait(waiting[0]);

signal(waiting[1]);

} else {

if (id < N-1)

wait(waiting[id]);

// Poslednji prolazi i signalizira pretposlednjem

signal(waiting[id-1]);

wait(waiting[id]);

// Tako se prenosi do 0, a onda se vraca na gore do poslednjeg

if (id < N-1)

signal(waiting[id+1]);

}

}

}

I)

program Barrier;

var sems : array[0..N-1] of semaphore;

procedure FirstProces

begin

while (true) begin

do\_smth;

for i:=1 to N-1 then wait(sems[i]);

for i:=1 to N-1 then signal(sems[0]);

end

end

procedure OtherProcess(var i : integer)

begin

while (true) begin

do\_smth;

signal(sems[i]);

wait(sems[0]);

end

end

begin

for i:=0 to N-1 then init(sems[i], 0)

cobegin

FirstProces()

OtherProcess(1)

..

OtherProcess(N)

coend

end

II)

program Barrier;

var sems : array[0..N-1] of semaphore;

procedure FirstProces

begin

while (true) begin

do\_smth;

wait(sem[0]);

signal(sem[1]);

end

end

procedure OtherProcess(var i : integer)

begin

while (true) begin

do\_smth;

if i + 1 < N then

wait(sems[i])

signal(sems[i-1]);

wait(sems[i]);

if i + 1 < N then

signal(sem[i+1])

end

end

begin

for i:=0 to N-1 then init(sems[i], 0)

cobegin

FirstProces()

OtherProcess(1)

..

OtherProcess(N)

coend

end

3. JUN(2019) Fine grain Ticket алгоритам реализован помоћу addAndGet операције. Уколико би addAndGet операција имала следећи ефекат: addAndGet(var, incr) : < var = var + incr; return(var); >, да ли је могуће направити Fine grain решење, полазећи од Coarse grain решења, и ако је могуће – направите га.

COURSE GRAINED

int tickets[N] = 0;

int nextTicket = 1, next = 1;

process CSi[0..N] {

while(true) {

< tickets[i] = nextTicket; nextTicket++ >

< await(next == tickets[i]) >

// critical section

< next++ >

}

}

FINE GRAINED

int tickets[N] = 0;

int nextTicket = 0, next = 1;

process CSi[0..N] {

while(true) {

tickets[i] = addAndGet(nextTicket, 1);

while(tickets[i] != next);

// critical section

next++

}

}

4. JUN(2019) Рачун у банци може да дели више корисника (The Savings Account Problem). Сваки корисник може да уплаћује и подиже новац са рачуна под условом да салдо на рачуну никада не буде негативан, као и да види тренутно стање рачуна. Уколико на рачуну нема довољно новца корисник треба да чека док неко други не уплати новац на тај рачун. Решити проблем користећи семафоре, не треба проверавати да ли корисник сме да приступи неком рачуну.

program TheSavingsAccountProblem

var balance : shared integer;

var mutex, waiting : semaphore;

var needBalance : shared integer;

procedure deposit(var val : integer)

begin

wait(mutex)

balance += val;

if needBalance > 0 && balance >= needBalance then

signal(waiting)

else signal(mutex)

end

procedure withdraw(var val : integer)

begin

wait(enter);

wait(mutex);

if(balance >= val) then

balance := balance + val;

else begin

needBalance := val;

signal(mutex);

wait(waiting);

balance := balance - val;

needBalance := 0;

end

signal(mutex);

signal(enter);

end

5. FEB(2019) Напишите решење коришћењем await исказа решење за ticket алгоритам. Ако не постоји fetch\_and\_add инструкција, већ само test\_and\_set, реализујте fine grain решење са запосленим чекањем за ticket алгоритам. Решење не треба да се бави проблемом инвалидације кеш меморија.

int next = 1, number = 1, tickets[n] = 0;

process CSi[0..n] {

while(true) {

<tickets[i] = number; number = number + 1>

<await(next == tickets[i])>

// critical section

<next = next + 1>

}

}

int next = 1, number = 1, tickets[n] = 0;

bool mutex = false;

process CSi[0..n] {

while(true) {

while(TS(mutex));

tickets[i] = number; number = number + 1

mutex = false;

while(next != tickets[i]);

// critical section

next = next + 1;

}

}

6. FEB(2019) Користећи расподељене бинарне семафоре и технику предаје штафетне палице, написати функцију zameni(vrednost), која за два узастопна позивајућа процеса мења њихове вредности, тј. први као повратну вредност треба да добије вредност другог позивајућег процеса, а други вредност првог; трећи добија вредност четвртог, а четврти трећег, итд.

program Swap;

var cnt, shared\_val : shared integer;

var mutex, value21, value12, enter2: semaphore;

procedure swap(var value : integer);

var new\_value : integer;

begin

// wait(enter2); NE TREBA

wait(mutex);

cnt := cnt + 1;

if cnt == 1 then begin

signal(mutex);

wait(value21);

new\_value := shared\_val;

shared\_val := value;

signal(value12);

end

else if cnt == 2 then

cnt := 0;

shared\_val := value;

signal(value21);

wait(signal12);

new\_value := shared\_val;

signal(mutex);

end

// signal(enter2);

return new\_value;

end

enter2 = 2, mutex = 1, value21 = 0, value12 = 0

7. JAN(2019) Решити проблем филозофа који ручавају (The Dining Philosophers) користећи расподељене бинарне семафоре и прослеђивање штафетне палице. Обезбедити да филозофи започињу ручак по редоследу доласка.

int state[N] = {thinking, ..., thinking};

sem sem[N] = 0;

void left(int i) { return (i-1+N)%N; }

void right(int i) { return (i+1)%N; }

void philosopher(int id) {

while (true) {

// THINKING

wait(mutex);

state[id] = hungry;

if (state[left(id)]!=eating && state[right(id)]!=eating) {

signal(mutex);

wait(sem[id]);

}

state[id] = eating;

signal(mutex);

// EATING

wait(mutex);

state[id] = thinking;

if (state[left(id)]==hungry && state[left(left(id))]!=eating)

signal(sem[left(id)]);

if (state[right(id)]==hungry && state[right(right(id))]!=eating)

signal(sem[right(id)]);

else

signal(mutex);

}

}

8. AVG(2019) Написати и објасните Fine grain Ticket алгоритам реализован помоћу FA операције. Уколико би уместо FA постојала операција Compare-And-Swap која би недељиво обављала (

CAS(a, b, c) :

< if (a == c)

{ c = b; return true;}

else { a = c; return false;}>);

да ли је могуће направити Fine grain решење и ако је могуће – направите га.

int next = 1, number = 1, tickets[N] = 0;

proccess CSi[0..n] {

tickets[i] = FA(number, 1);

while(tickets[i] != next);

// critical section

next++;

}

int next = 1, number = 1, tickets[N] = 0;

proccess CSi[0..n] {

int value = number;

do {

tickets[i] = value;

} while(!CAS(value, value + 1, number));

ALTERNATIVNO

//

int value = number;

while(!CAS(value, value + 1, number));

tickets[i] = value;

//

while(tickets[i] != next);

// critical section

next++;

}

9. AVG(2019) Посматра се проблем читалаца и писаца (Readers-Writers Problem). Користећи семафоре написати код за писце и за иницијализацију уколико је код за читаоце дат. Семафор rw се иницијализује на почетну вредност N (N>1), а поред овог семафора читаоци не користе другу синхронизацију.

procedure Reader(id : integer);

begin

wait(rw);

read\_db;

signal(rw)

end;

procedure Writer(id : integer);

begin

for i := 0 to N-1 do wait(rw);

write\_db;

for i := 0 to N-1 do signal(rw);

end;

begin

init(rw, N);

cobegin

Reader(0);

..

Reader(R);

Writer(0);

..

Writer(W);

coend

end;

10. SEP(2019) Потребно је имплементирати семафор s над којим су дефинисане операције signal и wait користећи атомску операцију Fetch and Add која има следећи ефекат: FA(var, incr) : <int tmp = var; var = var + incr; return(tmp);>. Претпоставити да су поред ове операције само још читање и упис у меморију атомске операције

int reserved = init, done = 0;

procedure signal() {

FA(reserved, 1);

}

procedure wait() {

int ticket = FA(done, 1);

while(ticket > reserved);

}

11. FEB(2018) Ticket алгоритам за критичну секцију имплементиран са Fetch and Add. Шта би се догодило када би се инкрементирање променљиве која одређује следећи процес радило у сваком процесу након (изван) критичне секције? Да ли би се у том случају коришћењем Fetch and Add за инкрементирање те променљиве изван критичне секције постигло задовољавајуће решење?

int next = 1, number = 1, tickets[N] = 0;

proccess CSi[0..n] {

tickets[i] = FA(number, 1);

while(tickets[i] != next);

// critical section

next++;

}

Za number je besmisleno, za next je po default takav.

12. JUN(2018) Синхронизација на баријери коришћењем координаторског процеса. Да ли су процеси који се синхронизују на баријери могли да бришу (ресетују) flag за стизање на баријеру, а да кôд буде исправан? Образложите.

Ne moze jer moze da se desi da Coordinator dodje ponovo do prevere, pre nego sto Worker ocisti arrive[i] i smatra da je zavrsio novu rundu iako nije.

Flagove ko ceka taj cisti.

process Worker[i] {

while(true) {

do\_smth;

arrive[i] = true;

<await (continue[i])>

continue[i] = false;

}

}

process Coordinator {

while(true) {

for [i := 0 to n]

await(arrive[i]);

arrive[i] = false;

for [i := 0 to n]

continue[i] = true;

}

}

13. JUN(2018) Посматра се један узак и релативно кратак пут кроз кањон, којим пролазе каубоји и Индијанци. Ако њиме пролазе само каубоји или само Индијанци, они ће бити фини и проћи једни поред других. Ако кроз клисуру желе да прођу и каубоји и Индијанци, примењује се „закон јачег“, тј. оних којих има више имају првенство пролаза кроз кањон. Треба обезбедити да једном добијено првенство пролаза не важи баш заувек – када се однос снага промени, треба забранити долазак нових особа које су до тада биле бројније, како би они други (сада бројнији) могли да добију право проласка кроз кањон. Коришћењем семафора, написати програм који решава овај проблем.

The Little Book of Semaphores, The Modus Hall Problem

14. K1(2015) Треба спровести лицитацију. На располагању су Test\_and\_Set и Fetch\_and\_Add инструкције. У дељеној променљивој izlicitirana\_vrednost се налази тренутна излицитирана вредност. Процеси који учествују у лицитацији треба запосленим чекањем да читају излицитирану вредност или да додају вредност на претходно прочитану вредност. Сматрати да одлучивање на основу прочитане вредности да ли треба подићи излицитирану вредност траје предуго да би се осталим процесима спречавао приступ и евентуална модификација променљиве излицитирана\_вредност. Није потребно писати процес који завршава лицитацију.

PROVERITI

int current\_value;

bool mutex = false;

process Licitation[i := 0 to N] {

while(true) {

int value = current\_value;

value = nextLicitation();

while(TS(mutex));

if(value > current\_value) {

current\_value = value;

}

mutex = false;

}

}

15. K1(2015) Посматра се скуп метода које служи за алокацију меморије (Memory Allocation Problem). У скупу постоје две методе за рад са меморијом капацитета N блокова: request(amount) и release(amount), где је amount позитиван цео број мањи или једнак N. Када неки процес позове методу request(amount) он се блокира све док не добије amount блокова меморије. Процес враћа amount блокова меморије позивом методе release(amount). Потребно је обезбедити да процес који је затражио мање меморије први буде опслужен (Shortest-Job-Next). Користећи семафоре решити овај проблем.

program MemoryAllocationProblem;

var capacity : shared integer;

var mutex : semaphore;

var queue : array[0..n] of semaphore;

procedure Request(id : integer, cap : integer)

begin

wait(mutex);

if (capacity < cap) then

begin

add (cap, id) to queue

signal(mutex);

wait(queue[id]);

end

capacity -= cap;

if q is empty then:

signal(mutex);

else:

id = pop shortest

if(can then) then

signal(queue[id]);

else signal(mutex);

end

procedure Release(cap : integer)

begin

wait(mutex);

capacity += cap;

if q is empty then:

signal(mutex);

else:

id = pop shortest

if(can then) then

signal(queue[id]);

else signal(mutex);

end

16. JUL(2015) Потребно је урадити синхронизацију на баријери која функционише на следећи начин: постоје два процеса координатора за по половину радних процеса (претпоставити паран број радних процеса) који треба да се синхронизују на баријери. Притом, та два процеса координатора прво закључују када су сви њихови процеси стигли до њихових интерних подбаријера и кроз поступак међусобне комуникације координатора утврђују када су сви радни процеси стигли до еквивалентне јединствене баријере. Након тога оба процеса координатора треба да дозволе радним процесима да прођу баријеру. Написати кôд радних процеса и оба процеса координатора.

process Worker[i := 0 to n] {

do\_smth;

arrive[i] = true;

<await(continue[i])>

continue[i] = false;

}

process Coordinator1 {

for [i := 0 to n/2]

<await(arrive[i]);

arrive[i] = false;

cord1\_finish = true;

<await(cord2\_finish)>

cord2\_finish = false;

for [i := 0 to n/2]

continue[i] = true;

}

process Coordinator2 {

for [i := n/2 to n]

<await(arrive[i]);

arrive[i] = false;

cord2\_finish = true;

<await(cord1\_finish)>

cord1\_finish = false;

for [i := n/2 to n]

continue[i] = true;

}

17. FEB1(2017) Tie breaker алгоритам за n процеса. Постоје 3 процеса и процес 1 уђе у прво стање и привремено престаје да се извршава. Од тог тренутка сви процеси добијају довољне временске интервале да стигну до тачке у којој не могу даље или улазе у критичну секцију, али секвенцијално добијају те временске интервале за извршавање. Редослед добијања временских интервала извршавања од је 3,2,1, ... (периодично). Који је редослед уласка у критичну секцију?

0 1 2 3

A X

B

C

0 1 2 3 CS

A X

B X

C X

0 1 2 3 CS

A X

B X

C X

0 1 2 3 CS

A X

B X

C X

Redom prolaze proces 1, pa 3, pa 2.

Napreduj ako nema niko bolji ili ako nisi poslednji usao u trenutni nivo.

18. FEB2(2018) Код Tie breaker алгоритма за n процеса се догодио следећи случај – приликом извршавања кода за улазак у критичну секцију, свих n процеса су ушли у стање 1 и ниједан још није ушао у стање 2. Одговорите на следећа питања: a) Да ли процес који је први ушао у стање 1 први улази у стање 2? б) Да ли процес који је први ушао у стање n-2 први улази у критичну секцију? в) Да ли процес који је последњи ушао у стање 1 може да буде трећи који улази у стање 2? г) Да ли процес који је последњи ушао у стање 1 може да буде први који улази у критичну секцију?

a) Ne, zavisi od zakazivanja ko bude sledeci.

b) Ne mora

c) Ne (osim ako je N = 3)

d) Ne

19. JUL2(2017) Flag синхронизација на баријери са процесом координатором. Објасните какве би последице на коректност рада баријере изазвала селидба операције arrive(i) = 0 у процесе раднике (worker) и опишите то за сва могућа места у кôду.

Ne moze jer moze da se desi da Coordinator dodje ponovo do prevere, pre nego sto Worker ocisti arrive[i] i smatra da je zavrsio novu rundu iako nije.

Flagove ko ceka taj cisti.

20. JUL2(2017) Користећи семафоре решити проблем берберина који спава (The Sleeping Barber Problem). Берберница се састоји од чекаонице са N столица и берберске столице на којој се људи брију. Уколико нема муштерија, брица спава. Уколико муштерија уђе у берберницу и све столице су заузете, муштерија не чека, већ одмах излази. Уколико је берберин зaузет, а има слободних столица, муштерија седа и чека да дође на ред. Уколико берберин спава, муштерија га буди.

// Ima resenje i u knjizi The Little Book of Semaphores

program TheSleepingBarberProblem

var waiting : shared integer;

var mutex, shaved : semaphore;

var waiting : array[0..n] of semaphore;

var queue : Queue;

var sleeping : bool;

procedure Barber

begin

while(true) begin

wait(mutex);

if queue is Empty then begin

sleeping := true;

signal(mutex);

wait(wakeup);

sleeping := false;

end

waiting := waiting - 1;

client\_id := queue.pop();

signal(waiting[client\_id]);

signal(mutex);

trim;

signal(shaved);

end

end

procedure Client(id : integer)

begin

wait(mutex);

if waiting == N then begin

signal(mutex);

return;

end

waiting := waiting + 1

queue.push(id);

if sleeping then

signal(wakeup);

else

signal(mutex);

wait(waiting[id]);

wait(shaved);

end

21. K1(2017)

Посматра се шпил од 24 карте, подељене у 4 боје, са по 6 различитих бројева. Игру играју 4 играча, који седе за округлим столом и сваки од њих иницијално држи по 4 карте. Између два суседна играча се налази гомила са картама, која може у неком тренутку бити празна, а иницијално садржи 2 карте. Игра се завршава када неки играч објави да има све 4 карте истог броја, у различитим бојама, и тада сви играчи прекидају игру. Сваки играч, док год нема 4 исте и нико није објавио да је победник, избацује једну карту из своје руке и ставља је на гомилу са своје леве стране, потом узима једну карту са врха из гомиле са своје десне стране. Претпоставити да су играчима иницијално подељене карте на случајан начин. Користећи семафоре, написати изглед процедуре за једног играча. Поред играча, не постоји ниједан други процес.

program Cards;

var cards : array[1..n] of semaphore;

var cardMutex: array[1..n] of semaphore

var winMutex : semaphore;

var winner : shared integer;

procedure Player(var i: integer) begin

while(true) begin

if win begin

wait(winMutex);

if (winner == 0) then winner := i;

signal(winMutex);

end

wait(winMutex);

if (winner != 0) begin

signal(winMutex);

break;

end

signal(winMutex);

pick\_one\_card;

wait(cardMutex[next(i)]);

put\_card;

signal(cardMutex[next(i)]);

signal(cards[next(i)]);

wait(cards[prev(i)]);

wait(cardMutex[prev[i]])

get\_card

signal(cardMutex[prev(i)];)

end

end

begin

for i := 1 to n do init(2, cards);

for i := 1 to n do init(1, cardMutex);

init(1, winMutex);

winner := 0

cobegin

Player(0)

..

Player(N)

coend

end

22. K1(2017)

У Bakery алгоритму за N процеса, број процеса је 3. Напишите fine grain Bakery алгоритам. Сви процеси су распоредом извршавања иницијално добили вредности 1 за сопствене променљиве за редослед (turn[i]) и сви су прочитали све вредности turn[i] из осталих процеса у покушају да одреде нову сопствену вредност turn[i] и свугде прочитали вредности 1, али нису још одредили нову вредност turn[i]. Опишите разрешење и редослед уласка процеса у критичну секцију под условом да је од тог тренутка распоред извршавања процес 2, процес 1, процес 3, процес 2, процес 1, процес 3, …, а сваки од њих добије довољно времена да изврши бар неколико итерација програмске петље и кôд у критичној секцији.

process CS[id := 0 to 3] {

turn[id] = 1;

int maximum = 0;

for(int i = 0; i < 3; i++) {

if(id != i) {

maximum = turn[i];

}

}

turn[id] = maximum + 1;

for(int i = 0; i < 3; i++) {

if(id != i) {

while(turn[i] != 0 && (turn[i], i) < (turn[id], id)) skip;

}

}

// critical section

turn[id] = 0;

}

Koji god da je redosled, prvo ce treci pa drugi pa prvi, jer ce svima biti turn = 2, pa ce se porediti po IDijevima.

23. OCT(2017) Постоје три врсте операција које се могу обављати на једноструко уланчаном листом (The Search-Insert-Delete problem): претраживање, убацивање и брисање. Претраживање само прегледа листу, тако да се може дозволити да више процеса претражују у паралели. Убацивање додаје нови елемент на крај листе. Процеси који убацују су међусобно искључиви, али се убацивање може радити у паралели са произвољним бројем претраживања. Може се брисати елемент са било које позиције у листи. У једном тренутку може само један процес који брише да приступа листи. Тај процес има ексклузивни приступ листи. Користећи семафоре решити овај проблем.

// Postoji resenje u The Little Book of Semaphores

program SearchInsertDeleteProblem

var nS, nI, nD, waitingS, waitingI, waitingD : integer;

var mutex, search, insert, delete: semaphore;

// nD = 0;

procedure Search(val : integer)

begin

wait(mutex);

if (nD == 0) begin

waitingS += 1

signal(mutex);

wait(search);

waitingS -= 1;

end

nS := nS + 1

signal(mutex);

// search

wait(mutex);

nS := nS - 1

SIGNAL;

end

// nD == 0 && nI == 0;

procedure Insert(val : integer)

wait(mutex);

if (nD == 0 && nI == 0) begin

waitingI += 1

signal(mutex);

wait(insert);

waitingI -= 1;

end

nI := nI + 1

signal(mutex);

// insert

wait(mutex);

nI := nI - 1;

SIGNAL;

begin

end

// nS == 0 && nD == 0;

procedure Delete(pos : integer)

begin

wait(mutex);

if (nS == 0 && nD == 0) {

waitingD += 1

signal(mutex);

wait(delete);

waitingD -= 1;

}

nD := nD + 1

signal(mutex);

// delete

wait(mutex);

nD := nD - 1;

SIGNAL;

end

SIGNAL :

if(nD == 0 && waitingS > 0)signal(search)

else if(nd == 0 && nI == 0 && waitingI > 0)signal(insert)

else if(nS == 0 && nD == 0 && waitingD > 0)signal(delete);

else signal(mutex);

24. K1(2020) 1.

Node n = new Node(false);

void process() {

Node node = new Node(true);

<Node prev = n; n = node;>

<await(!prev.locked);>

// KRITICNA SEKCIJA

<node.locked = false;>

// NEKRITICNA SEKCIJA

}

//Fine grained

Node n = new Node(false);

void process() {

Node node = new Node(true);

Node prev = node;

SWAP(prev, n);

while (prev.locked) skip;

// KRITICNA SEKCIJA

node.locked = false;

// NEKRITICNA SEKCIJA

}

/\*

CLH algoritam je pravican jer ce procesi dobijati pristup kriticnoj sekciji

u redosledu u kom su nadolazili.

\*/

25. K1(2020) 2.

// Nije tacno resenje, vise kao ideja neka

// Potrebno je napisati cameOutside, dodati promenljivu koja kaze da je rampa zauzeta jer vozilo trenutno ulazi, srediti promenu prioteta...

const int N=..., K=...;

sem sem[2\*N+1];

const int mutex = 2\*N;

int waitingIn=0, waitingOut=0, parked=0;

int inHead=0, inTail=0, outHead=0, outTail=0;

int status = OUT;

int cnt=0;

void car() {

while (true) {

bool canEnter = enter();

if (canEnter) {

// entering

cameInside();

// do something

exit();

// exiting

// pay();

// exiting

cameOutside();

}

}

}

void enter() {

wait(sem[mutex]);

if (parked + waitingIn - waitingOut + 1 > N) {

signal(sem[mutex]);

return false;

} else if (waitingIn>0 || waitingOut>0) {

waitingIn++;

int myPlace = inHead;

inHead = (inHead+1)%N;

signal(sem[mutex]);

wait(sem[myPlace]);

waitingIn--;

}

cnt++;

regulate();

return true;

}

void cameInside() {

wait(sem[mutex]);

parked++;

regulate();

}

void exit() {

wait(sem[mutex]);

if (waitingIn > 0 || waitingOut > 0) {

waitingOut++;

int myPlace = outHead+N;

outHead = (outHead+1)%N;

signal(sem[mutex]);

wait(sem[myPlace]);

waitingOut--;

}

parked--;

cnt++;

regulate();

}

void regulate() {

// vec drzim mutex trenutno, predajem ga ili otpustam

if (waitingIn==0 && waitingOut>0 && status=IN) {

status = OUT;

cnt = 0;

} else if (waitingOut==0 && waitingIn>0 && status=OUT) {

status = IN;

cnt = 0;

} else if (waitingOut>0 && waitingIn>0 && status==IN && cnt==K) {

status = OUT;

cnt = 0;

} else if (waitingOut>0 && waitingIn>0 && status==OUT && cnt==K) {

status = IN;

cnt = 0;

}

if (waitingIn>0 && status == IN) {

int k = inTail;

inTail = (inTail+1)%N;

signal(sem[k]);

} else if (waitingOut>0 && status == OUT) {

int k = outTail+N;

outTail = (outTail+1)%N;

signal(sem[k]);

} else

signal(sem[mutex]);

}