**PROMELA ASSIGNMENT**

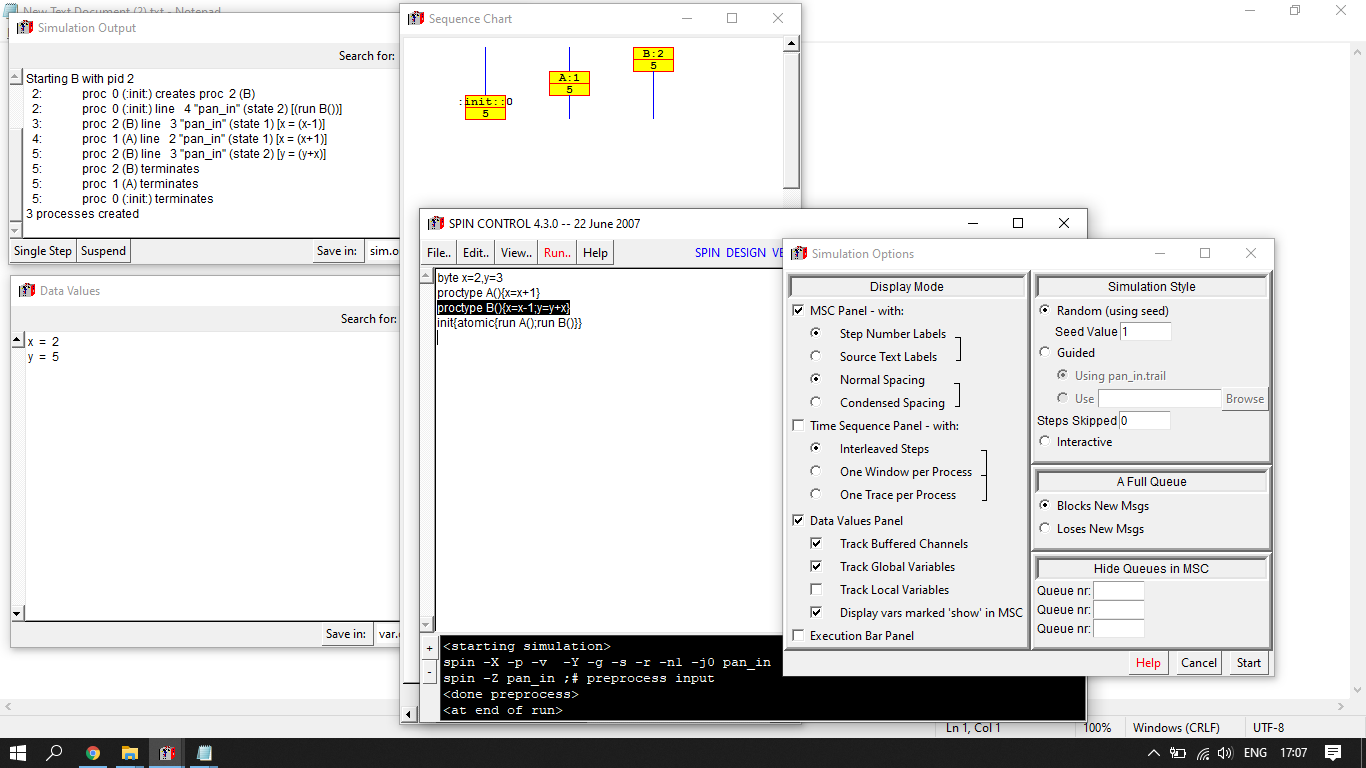
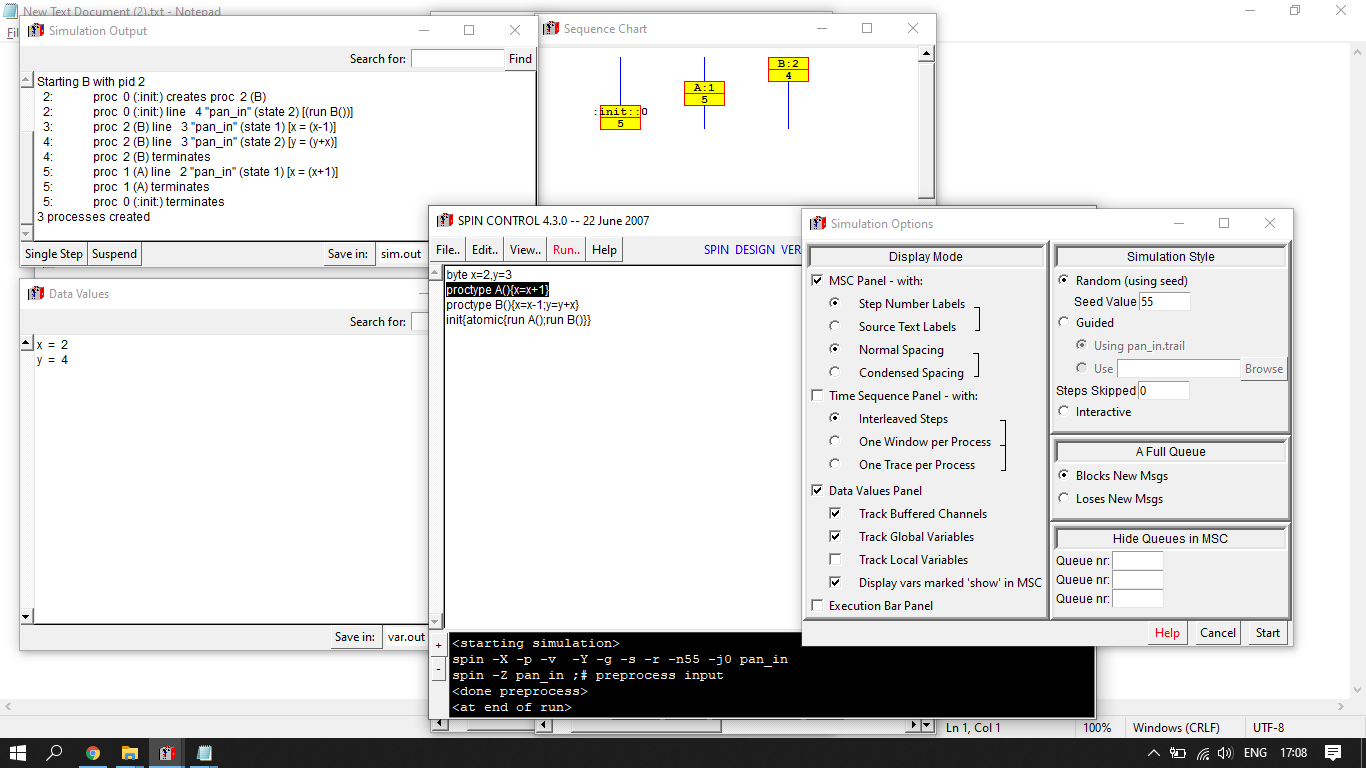
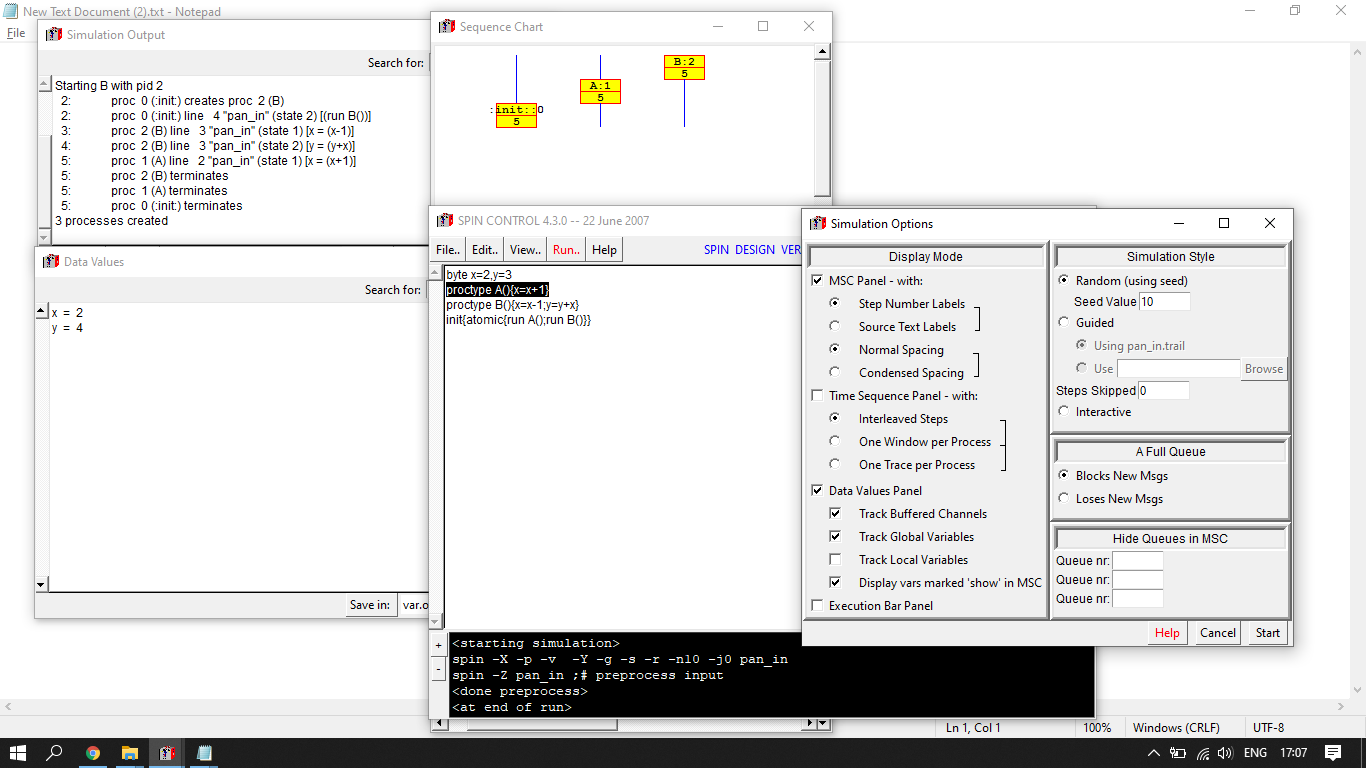
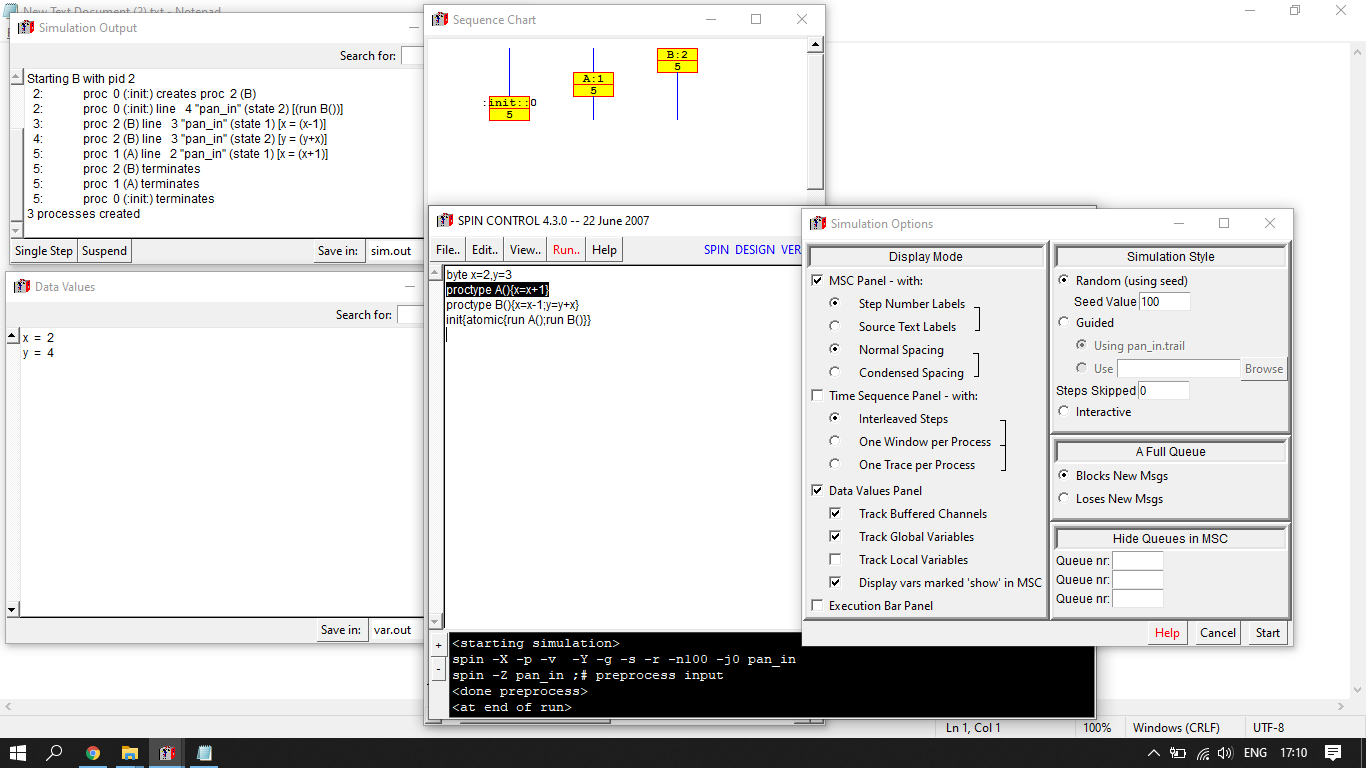
1. **1a.**

byte x=2,y=3

proctype A(){x=x+1}

proctype B(){x=x-1;y=y+x}

init{atomic{run A();run B()}}



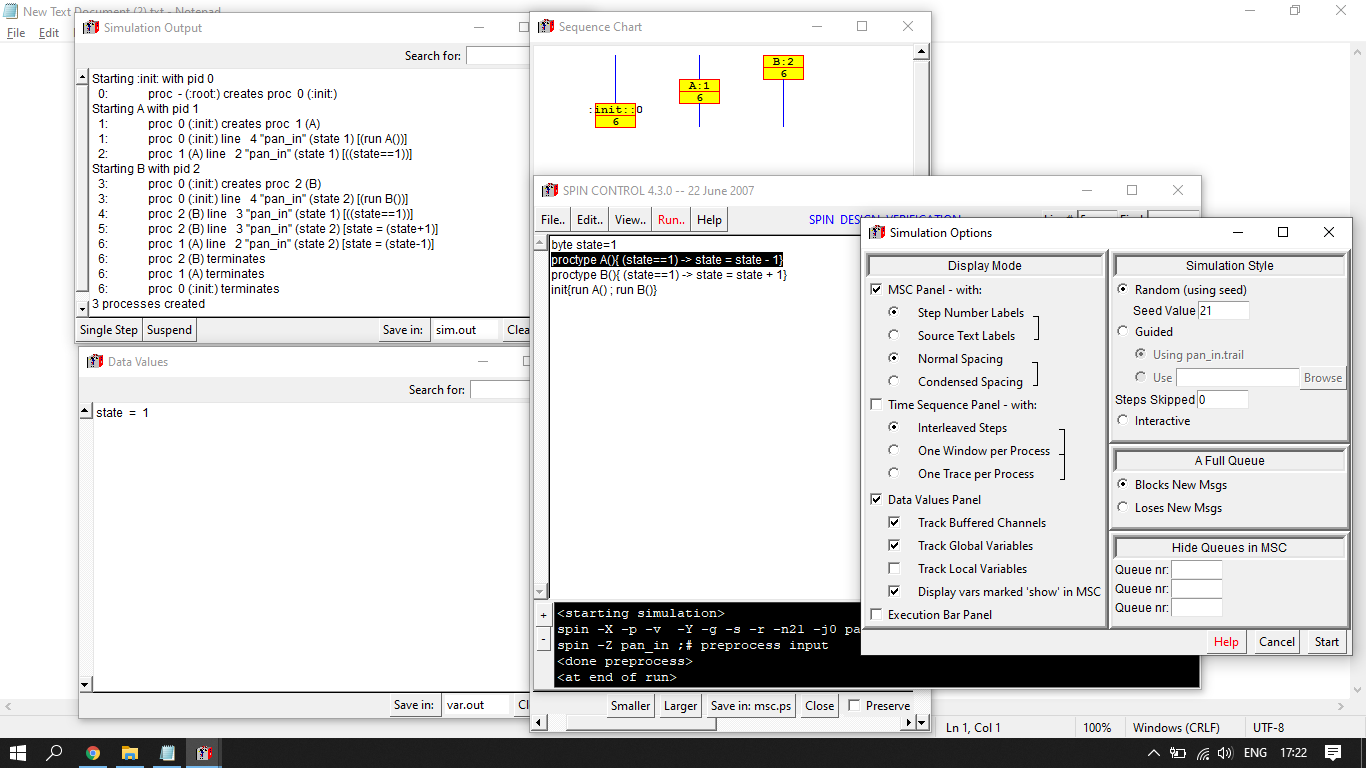
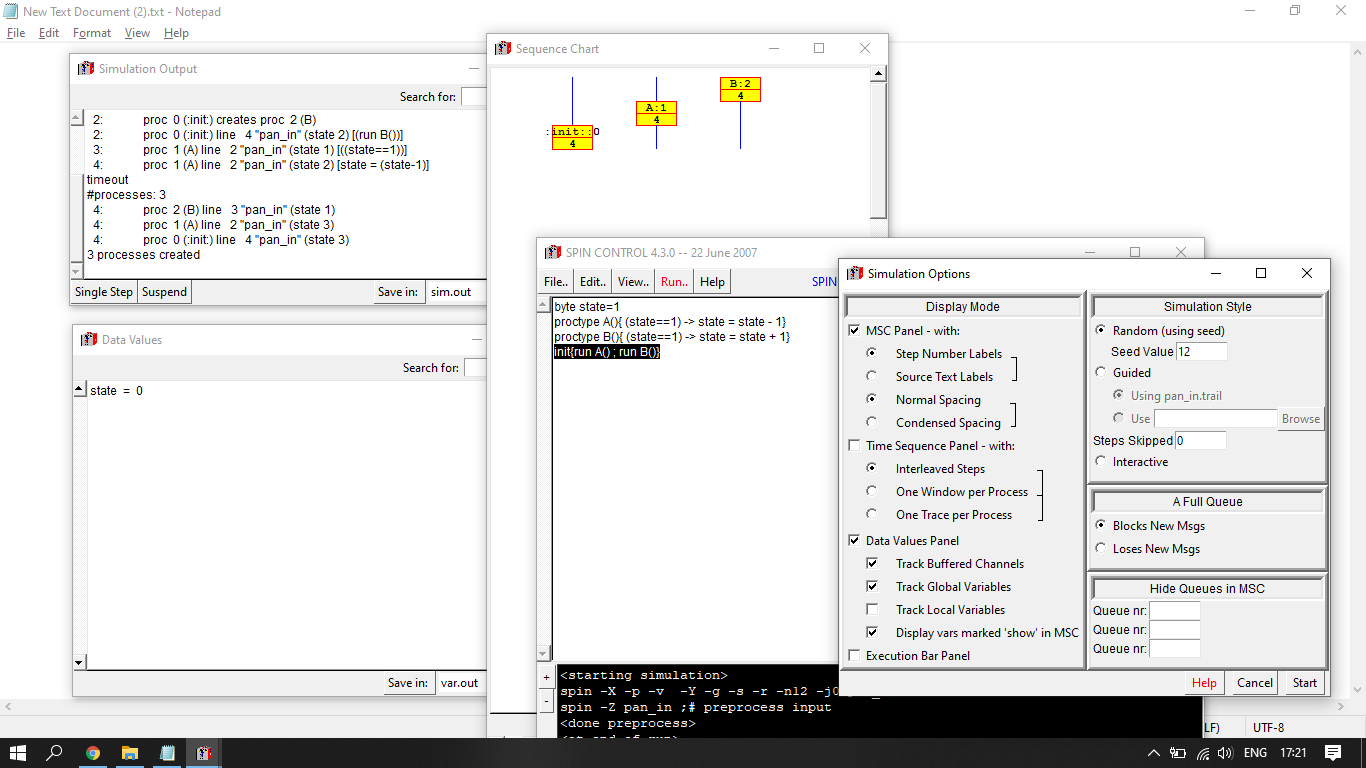
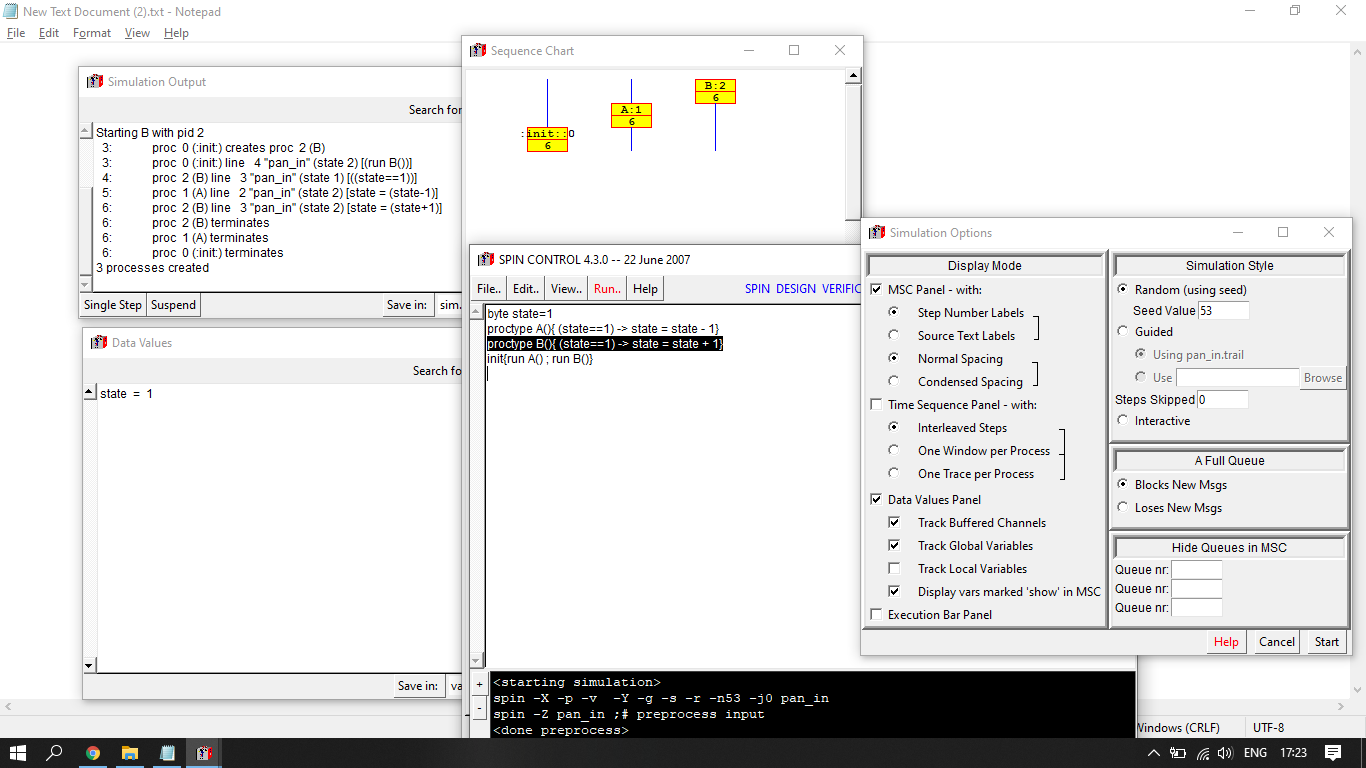
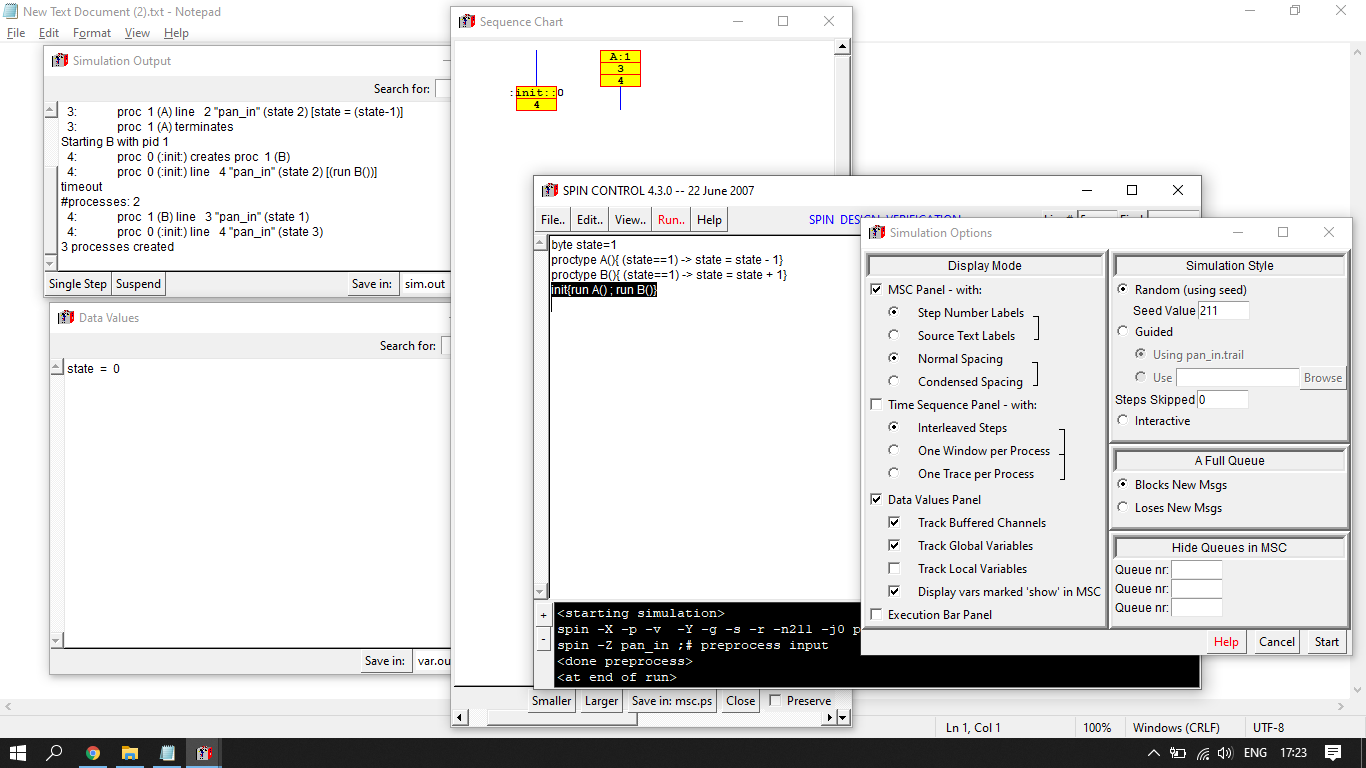
**1 b.**

byte state=1

proctype A(){ (state==1) -> state = state - 1}

proctype B(){ (state==1) -> state = state + 1}

init{run A() ; run B()}



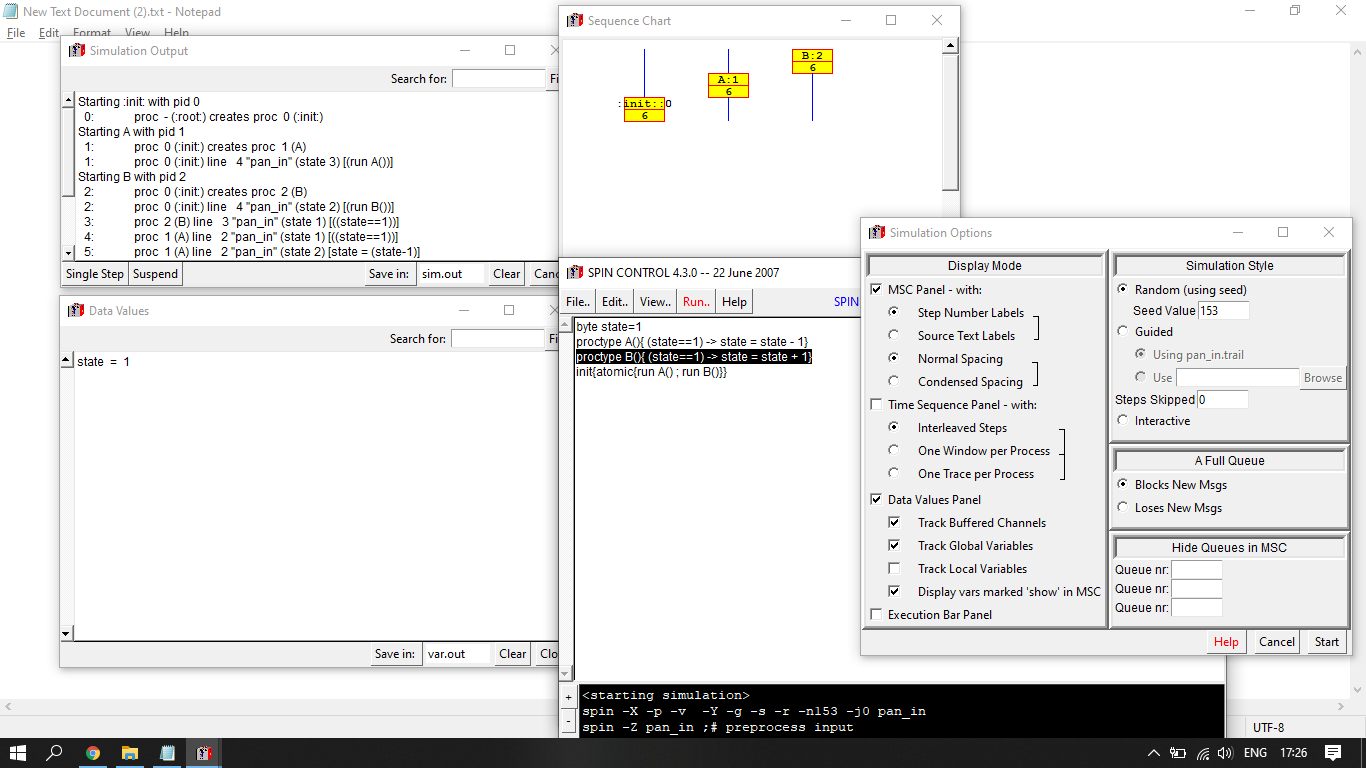
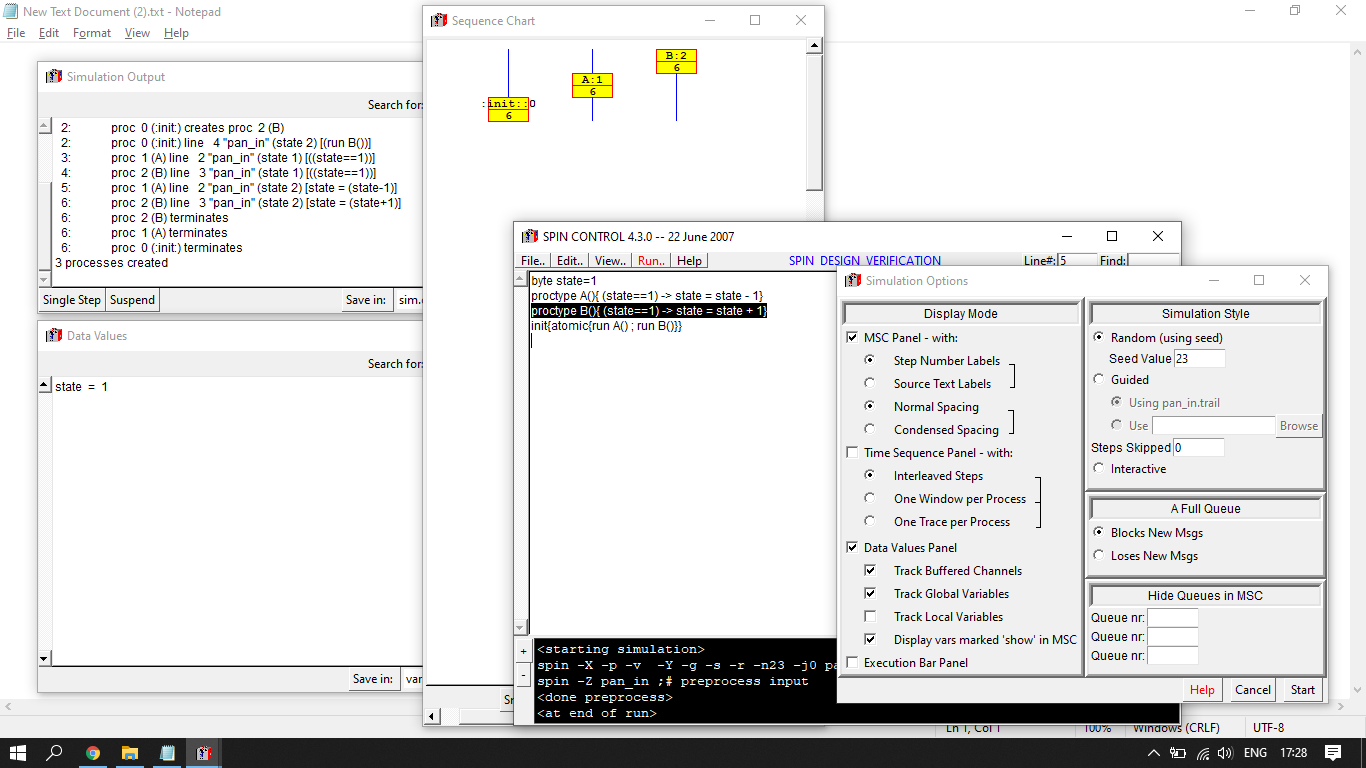
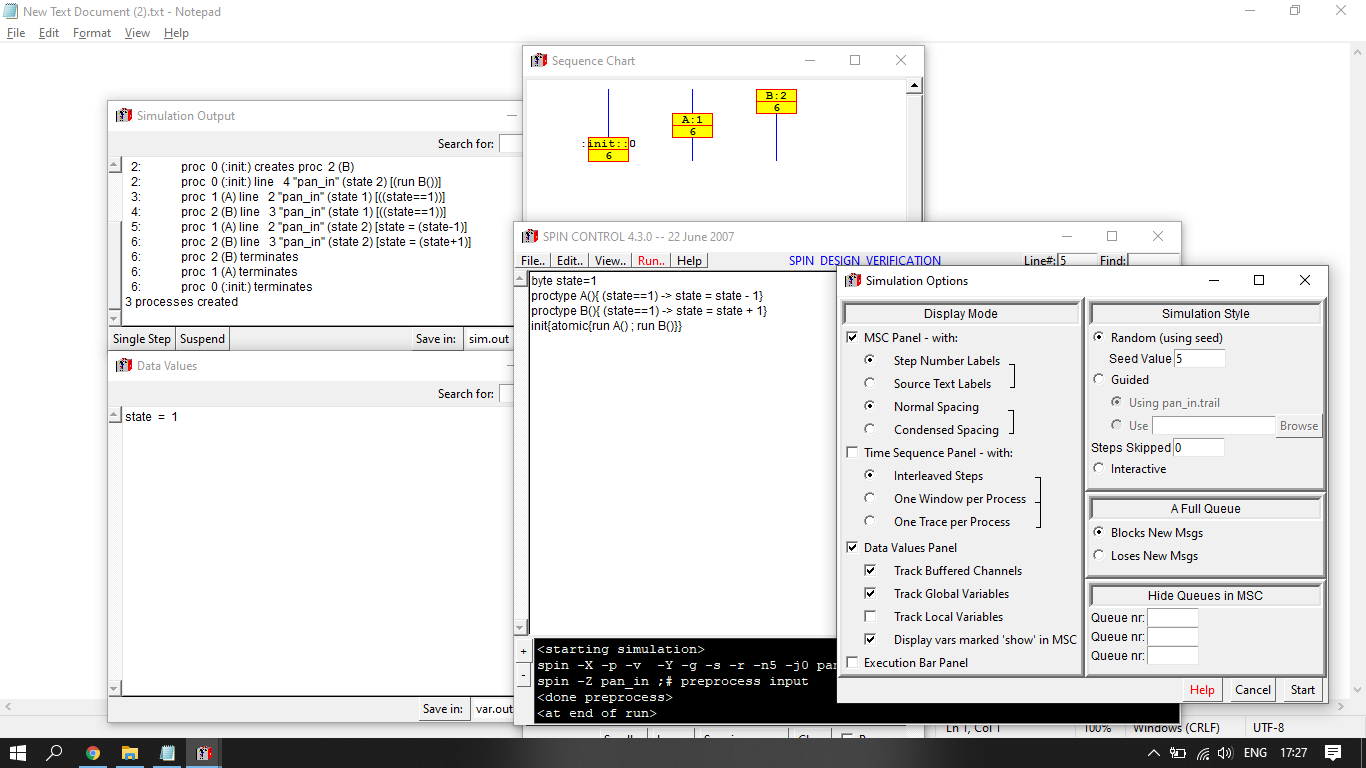
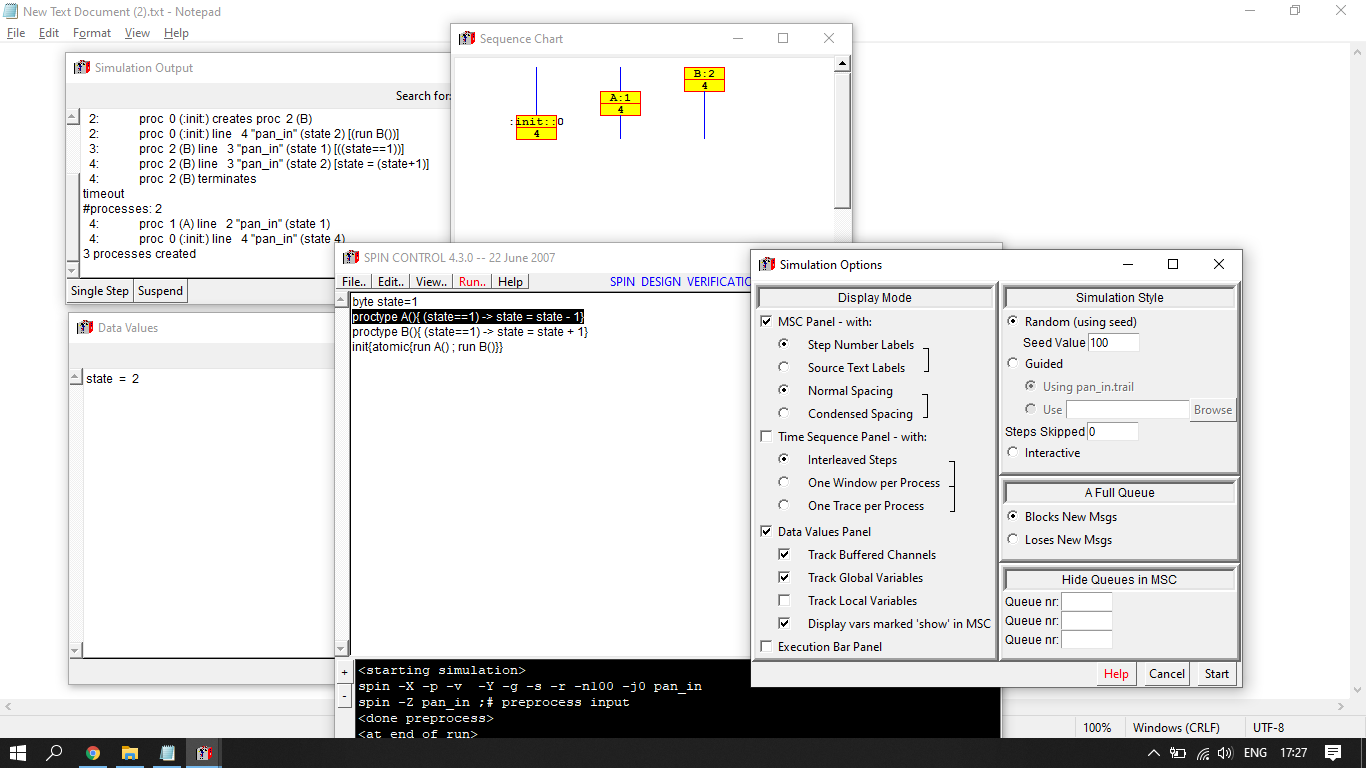
**1 c.**

byte state=1

proctype A(){ (state==1) -> state = state - 1}

proctype B(){ (state==1) -> state = state + 1}

init{atomic{run A() ; run B()}}



**1 d.**

byte state = 1

proctype A(){

atomic{ (state == 1) -> state = state + 1}

}

proctype B(){

atomic{ (state == 1) -> state = state - 1}

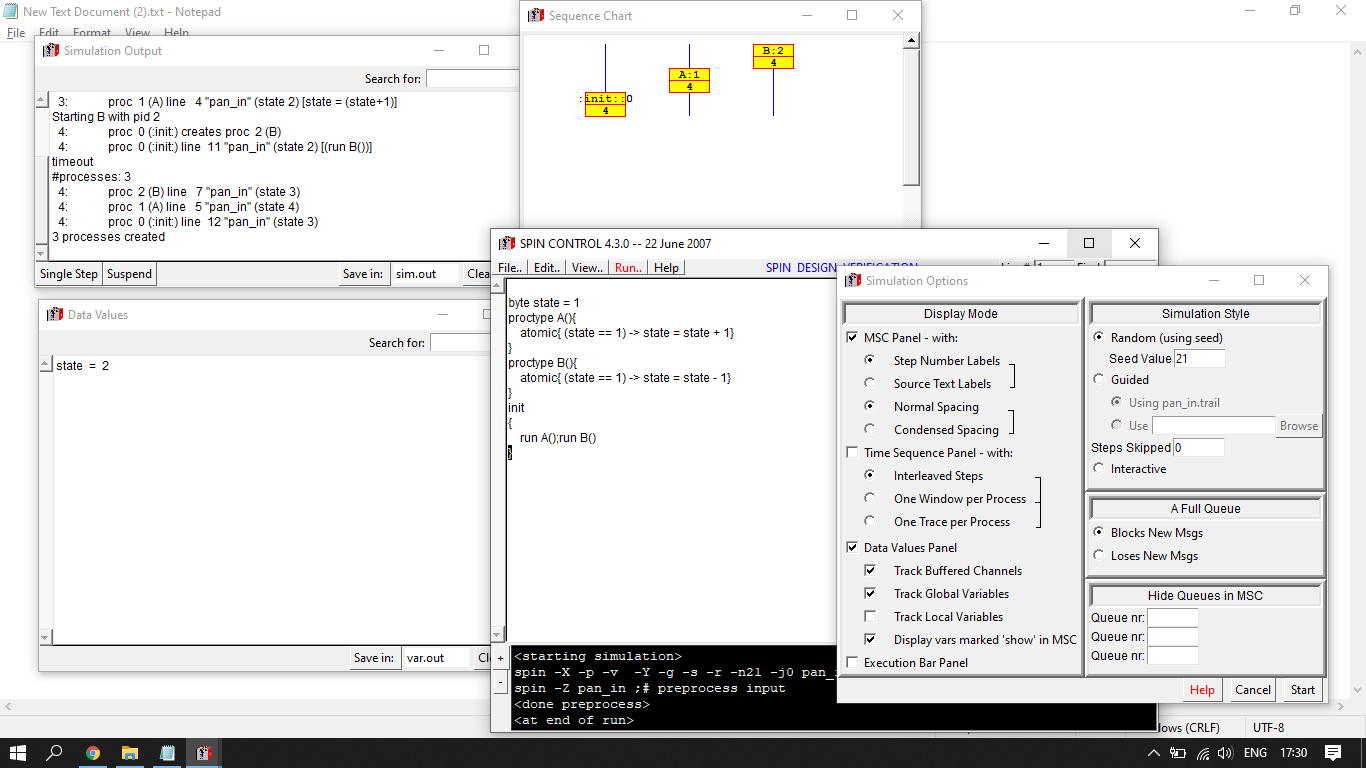
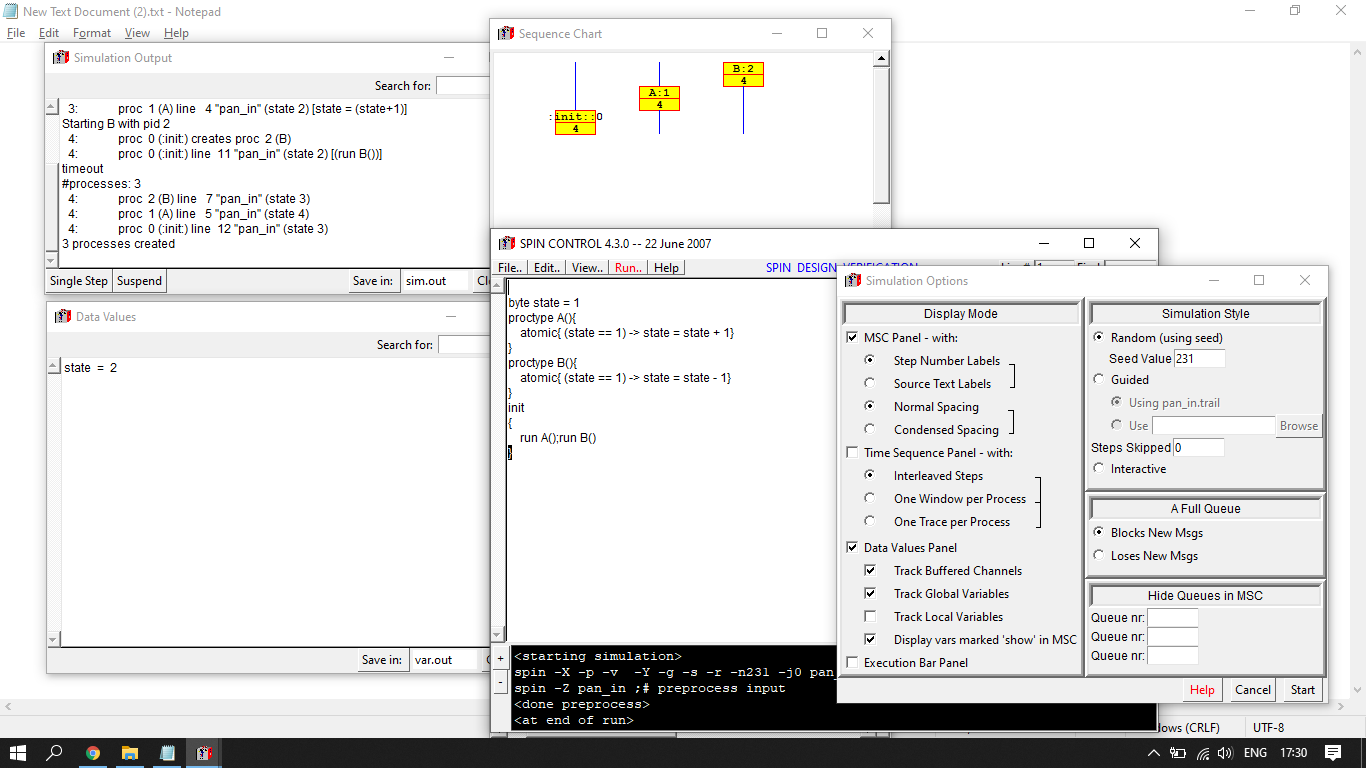
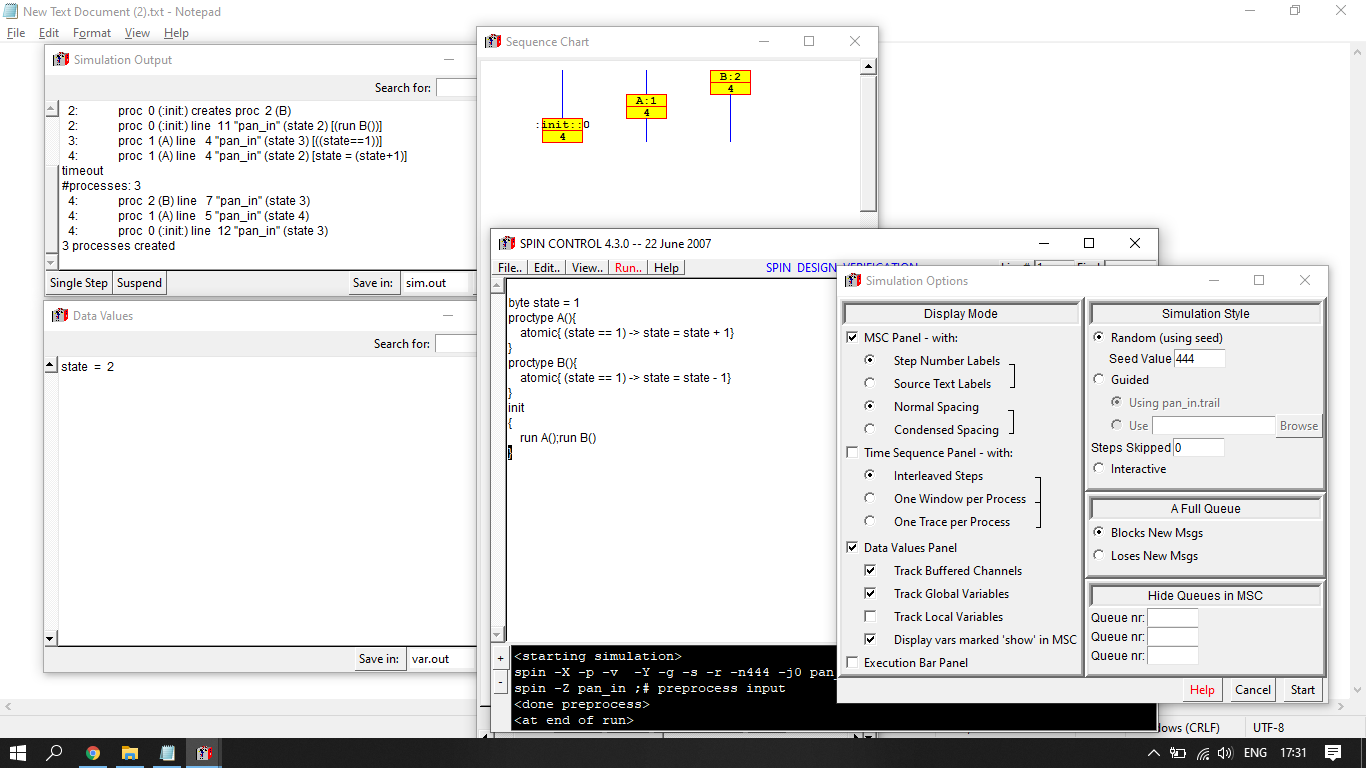
}

init

{

run A();run B()

}



**1 e.**

proctype hello(){

printf("Hello %d ",\_pid)

}

proctype world(){

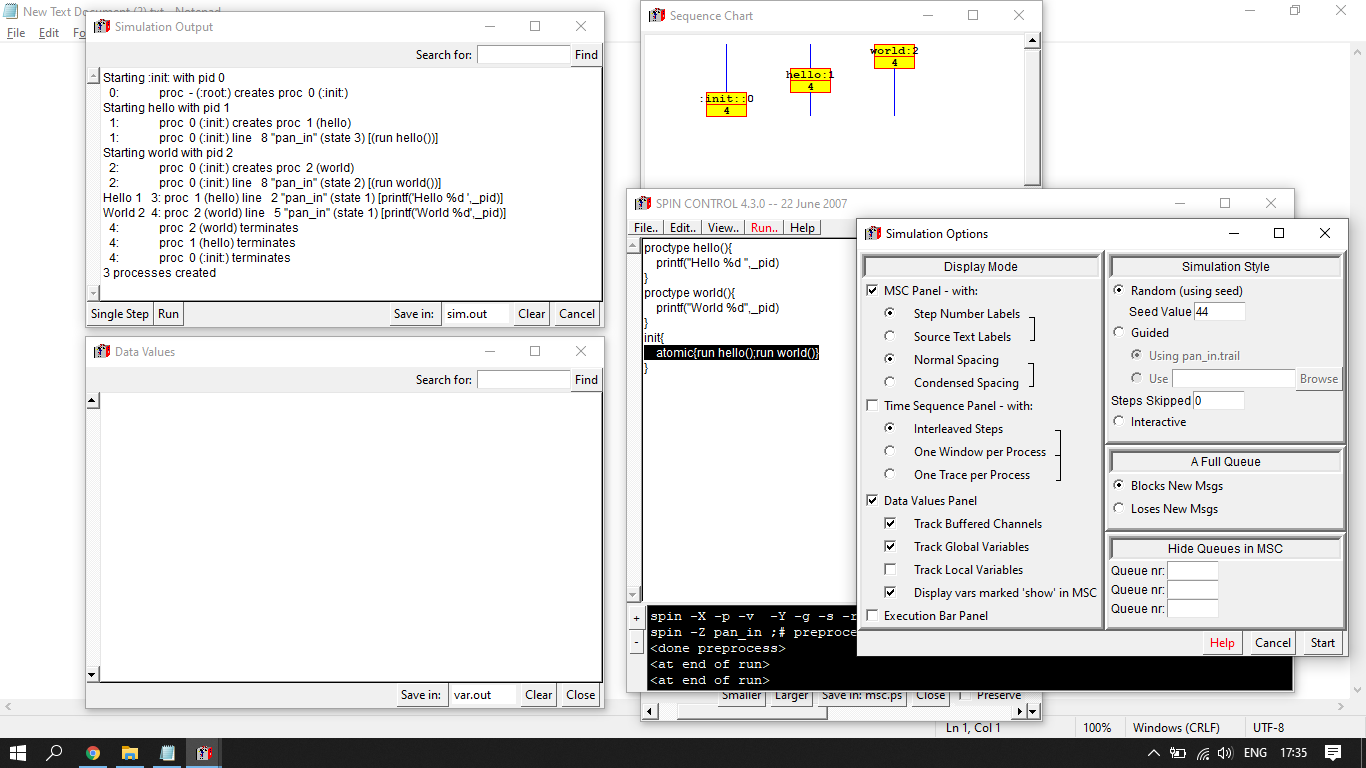
printf("World %d",\_pid)

}

init{

atomic{run hello();run world()}

}



**1 f.**

mtype = { written };

chan hello\_to\_control = [0] of { mtype };

chan control\_to\_world = [0] of { mtype };

proctype hello(chan x){

printf("Hello\n"); x!written;

}

proctype world(chan x){

x?written;

printf("World\n"); }

proctype control(chan x, y){

x?written;

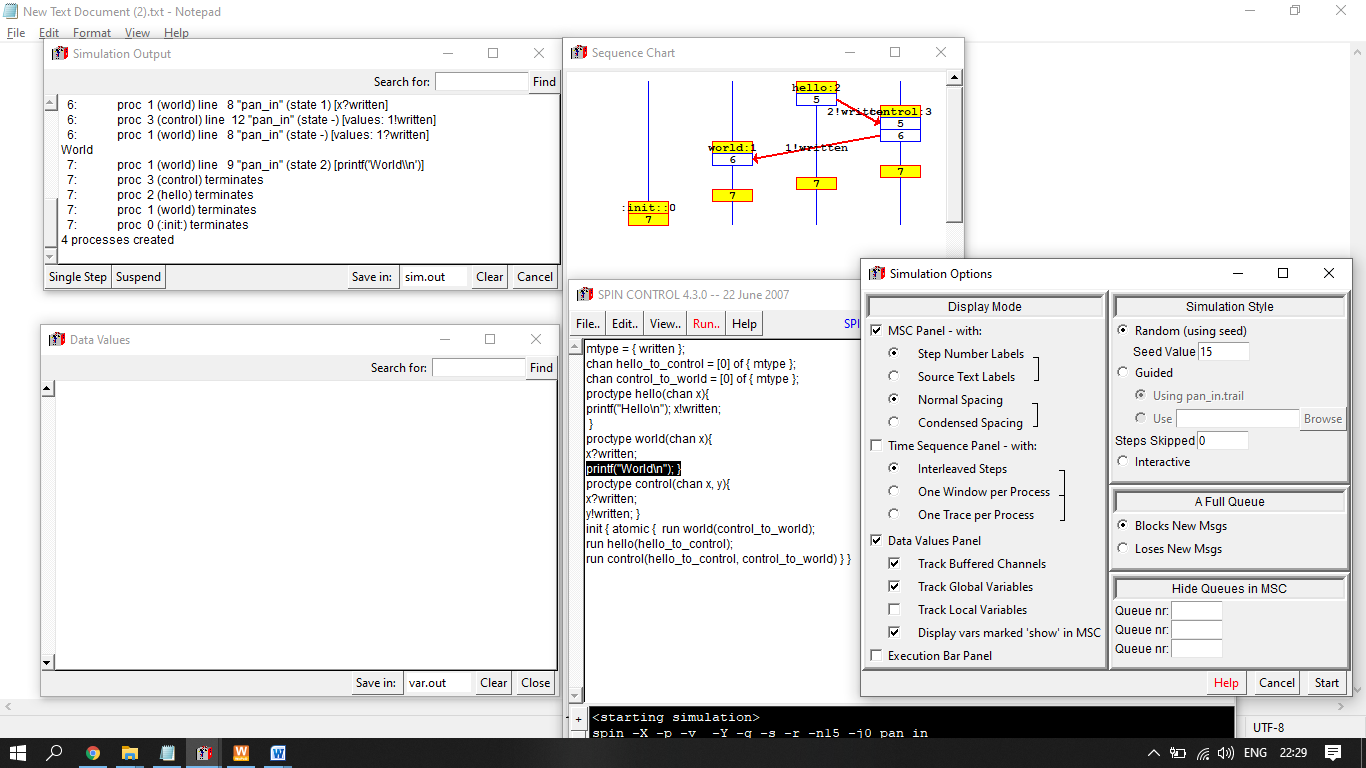
y!written; }

init { atomic { run world(control\_to\_world);

run hello(hello\_to\_control);

run control(hello\_to\_control, control\_to\_world) } }

..



**1 g.**

//Implementation of dekker's algorithm

#define true 1 false 0 Aturn 1 Bturn 0

bool Aruns;

bool Bruns;

bool t;

proctype A()

{

Aruns=true,

t=Bturn,

(Bruns == false || t == Aturn)

printf("A is running")

}

proctype B()

{

Bruns=true,

t=Aturn,

(Aruns == false || t == Bturn)

printf("B is running")

}

init{

run A();

run B();

}

1h>

byte in1, in2, a, b, quo, rem;

bit load = 0, done = 1;

proctype quo\_rem(){

do

::(load == 1) -> a =in1; b = in2;

quo = 0; rem = a; done =0;

::(load != 1) -> if

:: (rem >= b) -> rem = rem -b; quo = quo+1

::(b > rem) -> done = 1

fi

od;

}

proctype env(){

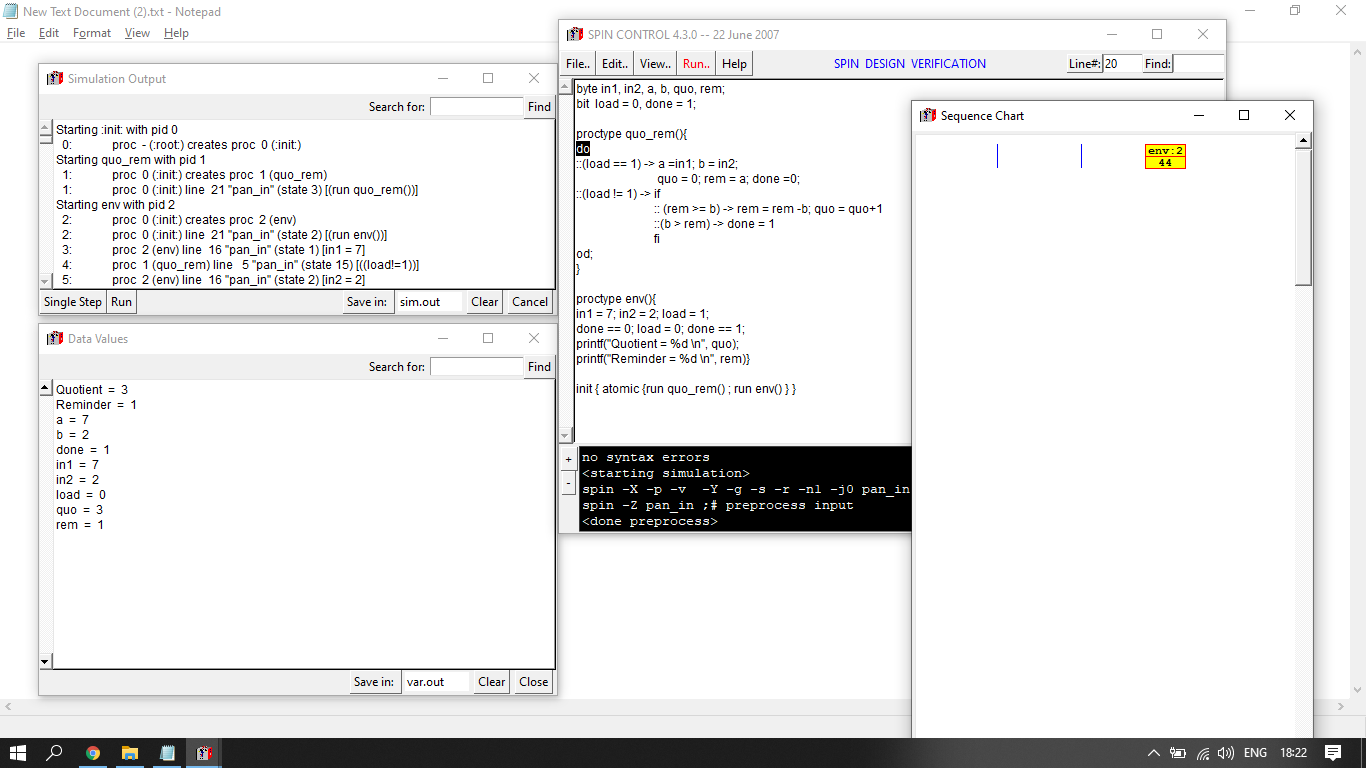
in1 = 7; in2 = 2; load = 1;

done == 0; load = 0; done == 1;

printf("Quotient = %d \n", quo);

printf("Reminder = %d \n", rem)}

init { atomic {run quo\_rem() ; run env() } }



**1 i.**

proctype fact(int n; chan p)

{ int result;

if

:: (n <= 1) -> p!1

:: (n >= 2) ->

chan child = [1] of { int };

run fact(n-1, child);

child?result;

p!n\*result

fi

}

init

{ int result;

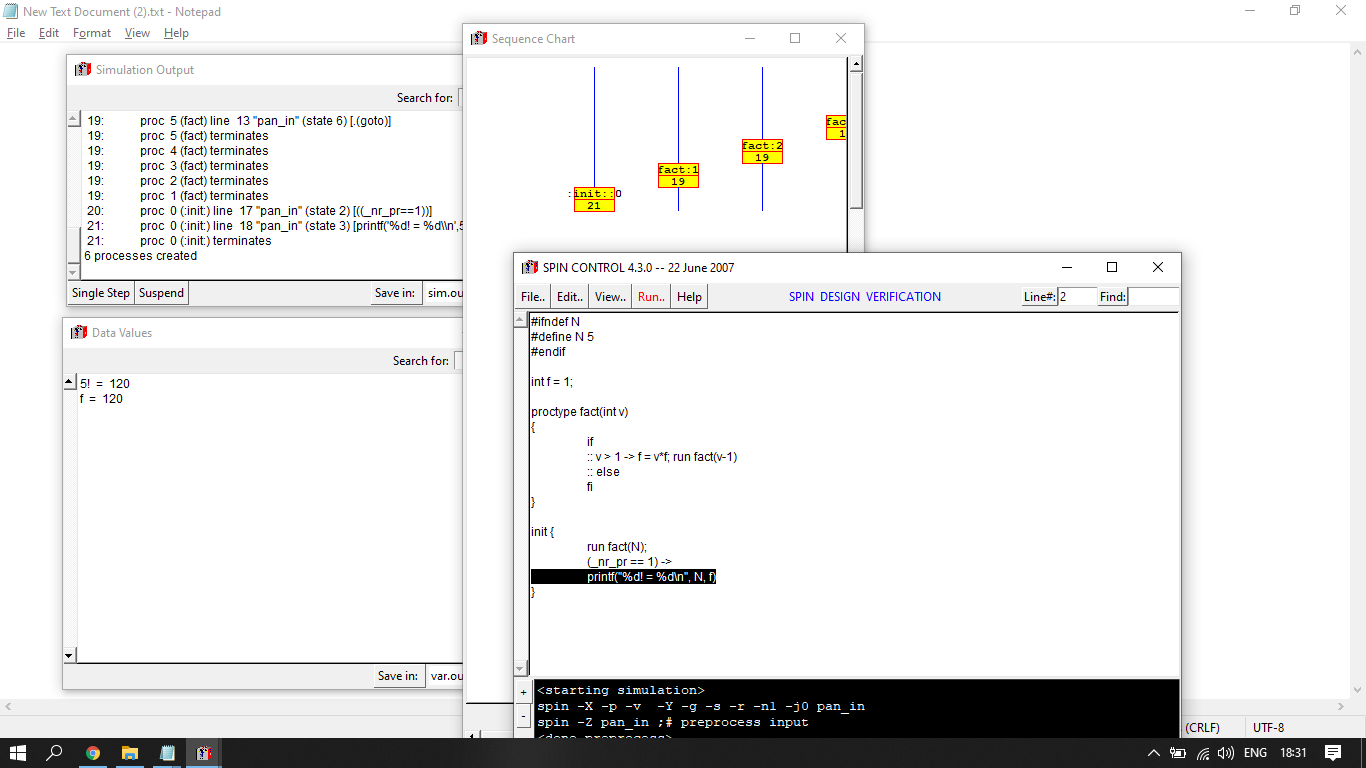
chan child = [1] of { int };

run fact(12, child);

child?result;

printf("result: %d\n", result)

}



**1 j.**

byte type item = 10;

var buffer: item;

status: (empty, full) { initially, empty };

procedure produce( x: item );

{ called by the producer when an item is ready to pass to the consumer }

begin

while status \(!= empty do { nothing };

buffer := x;

status := full;

end { produce };

procedure consume( var x: item );

{ called by the consumer when an item from the producer is needed }

begin

while status \(!= full do { nothing };

x := buffer;

status := empty;

end { consume };

**2.**

// Three readers and one writer

// if a reader is in the critical section it must allow different readers to read

chan lock = [0] of {bit}

chan rmutex = [0] of {bit}

bit x[10]

int index;

int readcount=0;

proctype lock\_provider()

{

bit p=0,v;

do

:: rmutex ! p

:: lock ! p

:: rmutex ? v

:: rmutex ! p

:: lock ? v

:: rmutex ? v

od

}

proctype reader()

{

bit p,v=1;

rmutex ? p

readcount = readcount + 1;

readcount==1

lock ? p

rmutex ! v

rmutex ? p

readcount=readcount-1

readcount==0

lock ! v

rmutex ! v

//(lock ! v)

}

proctype writer()

{

bit p=0,v;

lock ? p

x[index]=1

index=(index+1)%10;

lock ! v

}

init

{

atomic{run writer() ; run reader() ; run reader() ; run reader()}}

**3.**

#define N 5

bit forks[N];

byte count\_eating;

init {

atomic {

byte i = 0;

do

::(i < N-1) ->

run philosopher(i);

i++;

::else ->

run reset\_philosopher(i);

break;

od;

}

}

proctype philosopher(byte id) {

thinking:

atomic {

forks[id] == 0; ->

forks[id] = 1;

}

choosing:

atomic {

forks[(id + 1)%N] == 0 ->

forks[(id + 1)%N] = 1;

count\_eating++;

};

eating:

d\_step {

count\_eating--;

forks[(id + 1)%N] = 0;

}

forks[id] = 0;

goto thinking;

}

proctype reset\_philosopher(byte id) {

thinking:

atomic {

forks[id] == 0; ->

forks[id] = 1;

}

choosing:

if

::atomic {

forks[(id + 1)%N] == 0 ->

forks[(id + 1)%N] = 1;

count\_eating++;

};

::atomic {

forks[(id + 1)%N] != 0 ->

forks[id] = 0;

}

goto thinking;

fi;

eating:

d\_step {

count\_eating--;

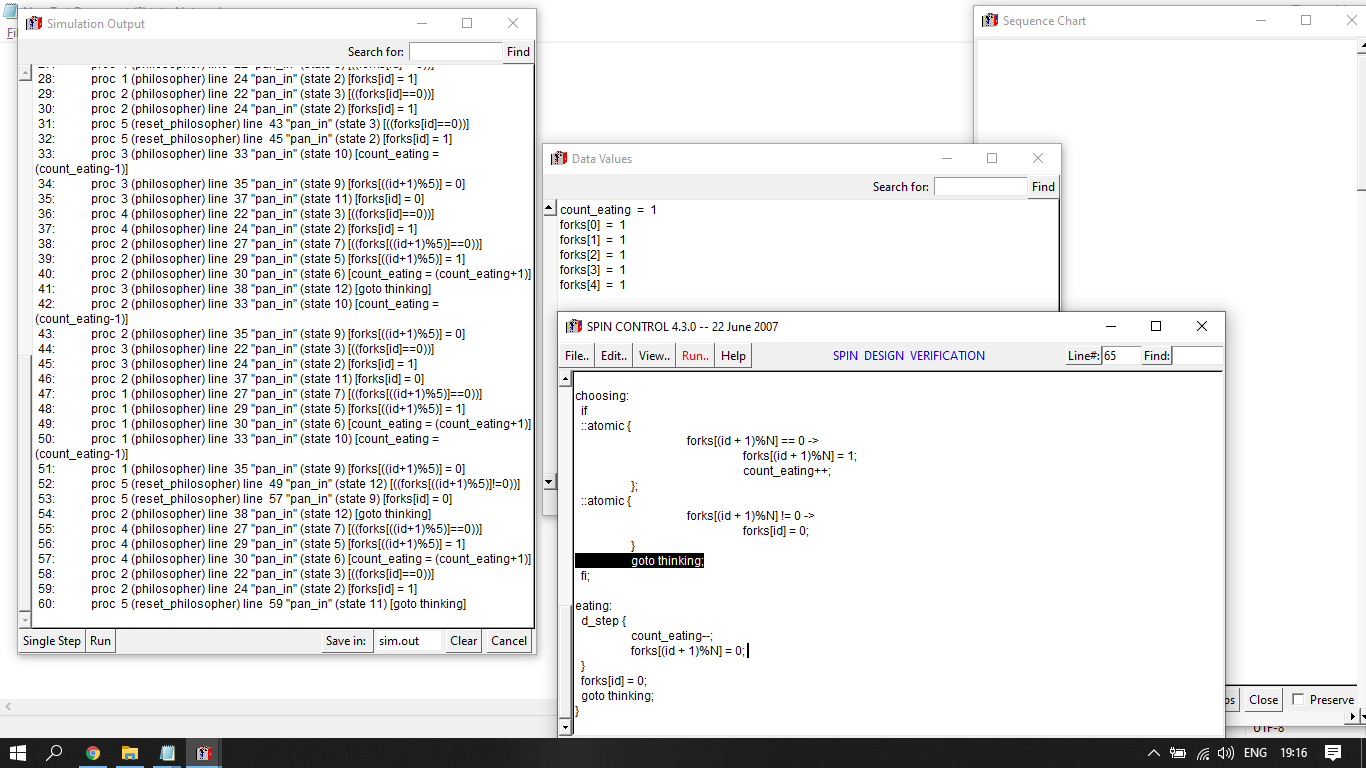
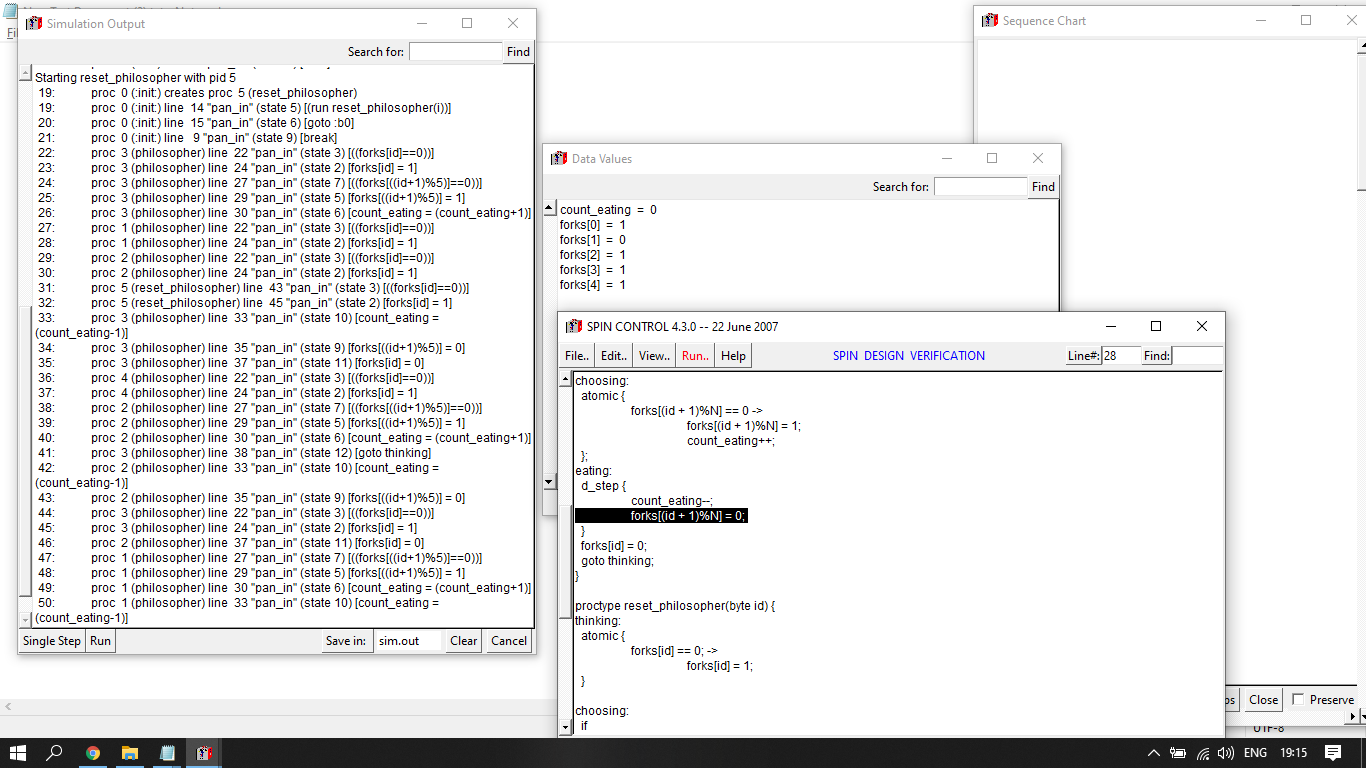
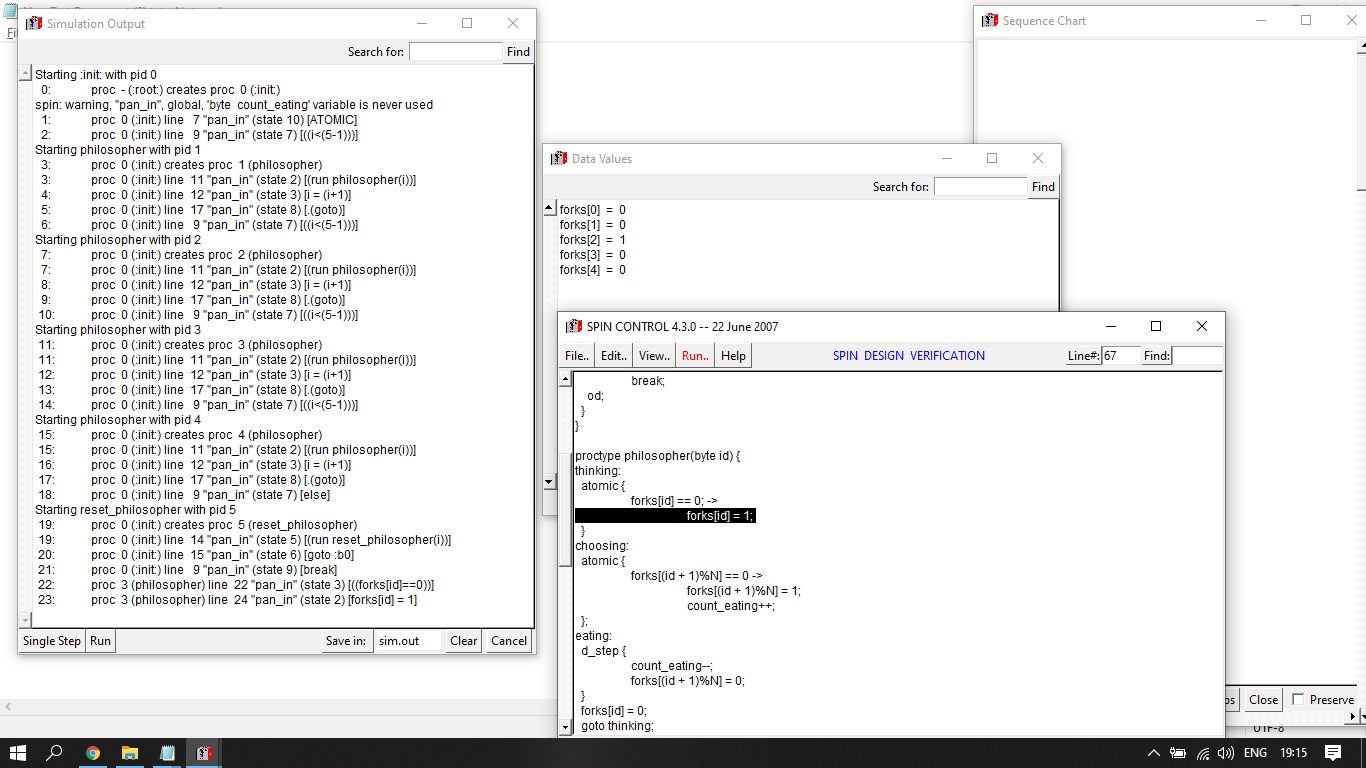
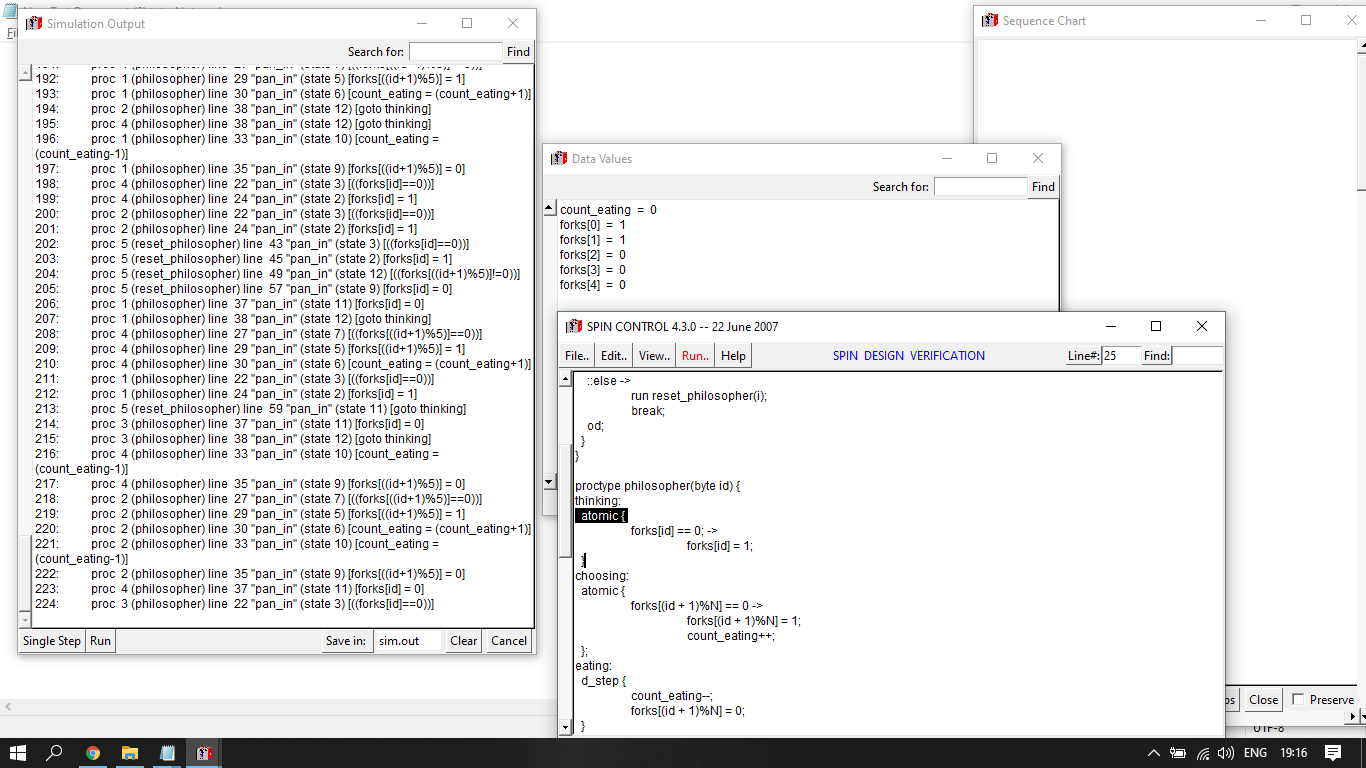
forks[(id + 1)%N] = 0;

}

forks[id] = 0;

goto thinking;

}



**4.**

#define train 1

chan BlockSecAB = [2] of { bit };

chan BlockSecBC = [2] of { bit };

proctype SignalA(chan out\_track)

{

do

:: out\_track!1

od

}

proctype SignalB(chan in\_track, out\_track)

{

do

:: in\_track?train;

out\_track!train

od

}

proctype SignalC(chan in\_track)

{

do

:: in\_track?train;

od

}

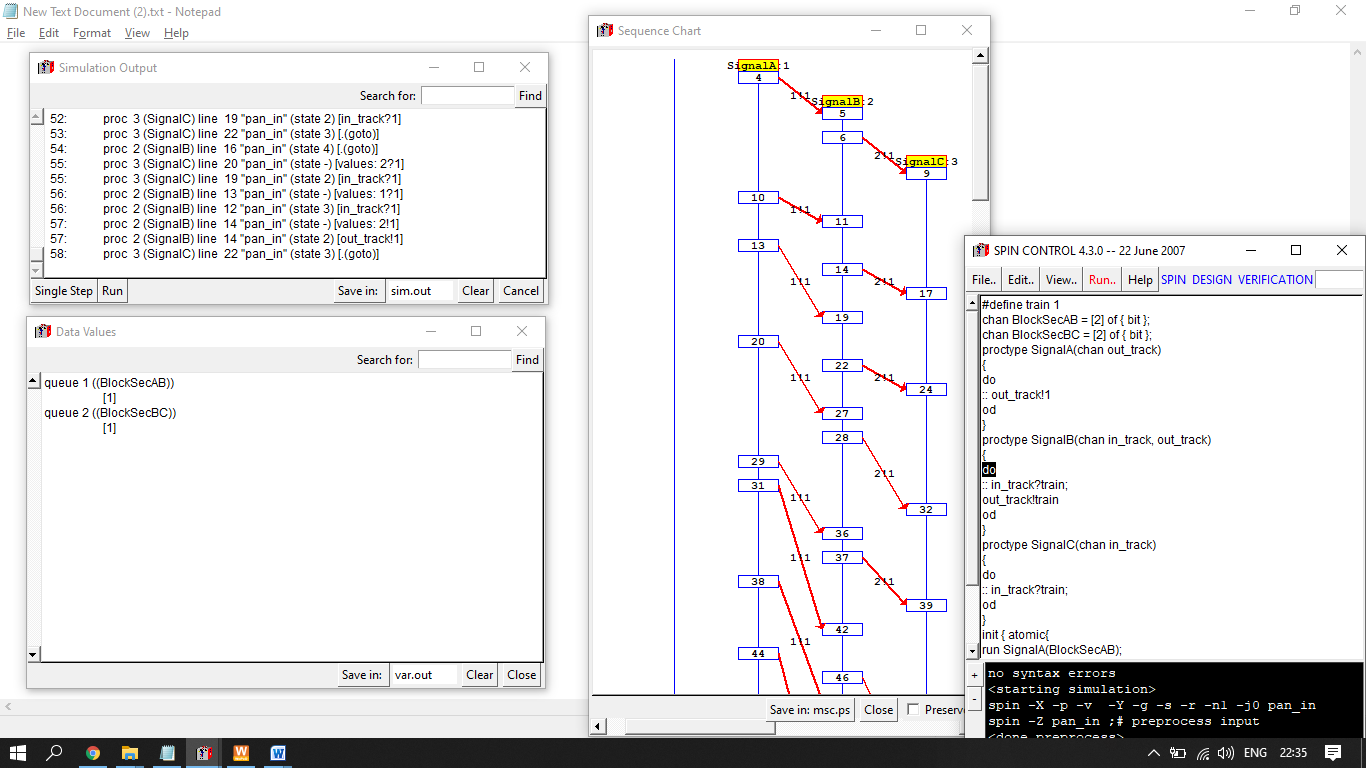
init { atomic{

run SignalA(BlockSecAB);

run SignalB(BlockSecAB, BlockSecBC);

run SignalC(BlockSecBC) }

}



**5.**

mtype = { coin5, coin10, milk, plain};

chan coin\_channel = [1] of { mtype };

chan choc\_channel = [1] of { mtype };

proctype customer(){

do

:: coin\_channel!coin5;

choc\_channel?milk;

:: coin\_channel!coin10;

choc\_channel?plain;

od

}

proctype vender(){

do

:: coin\_channel?coin5;

choc\_channel!milk;

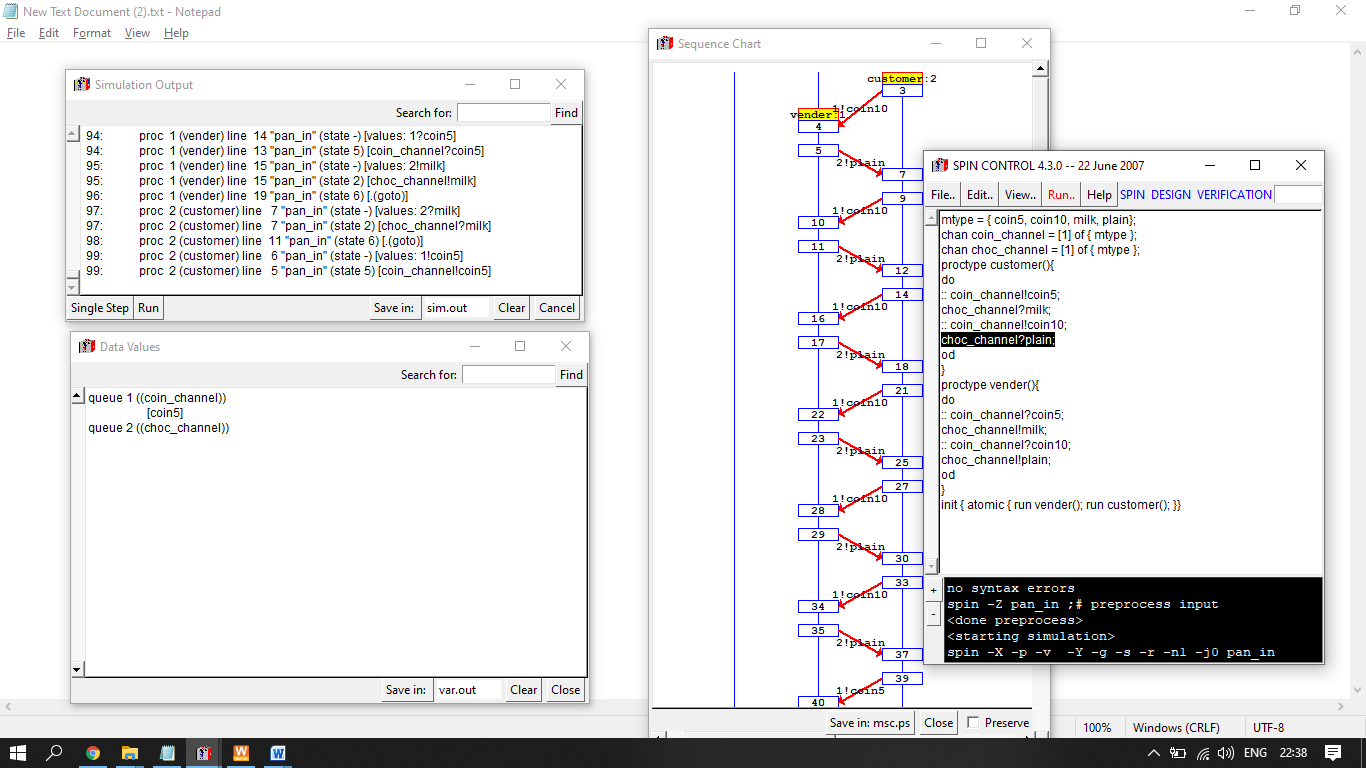
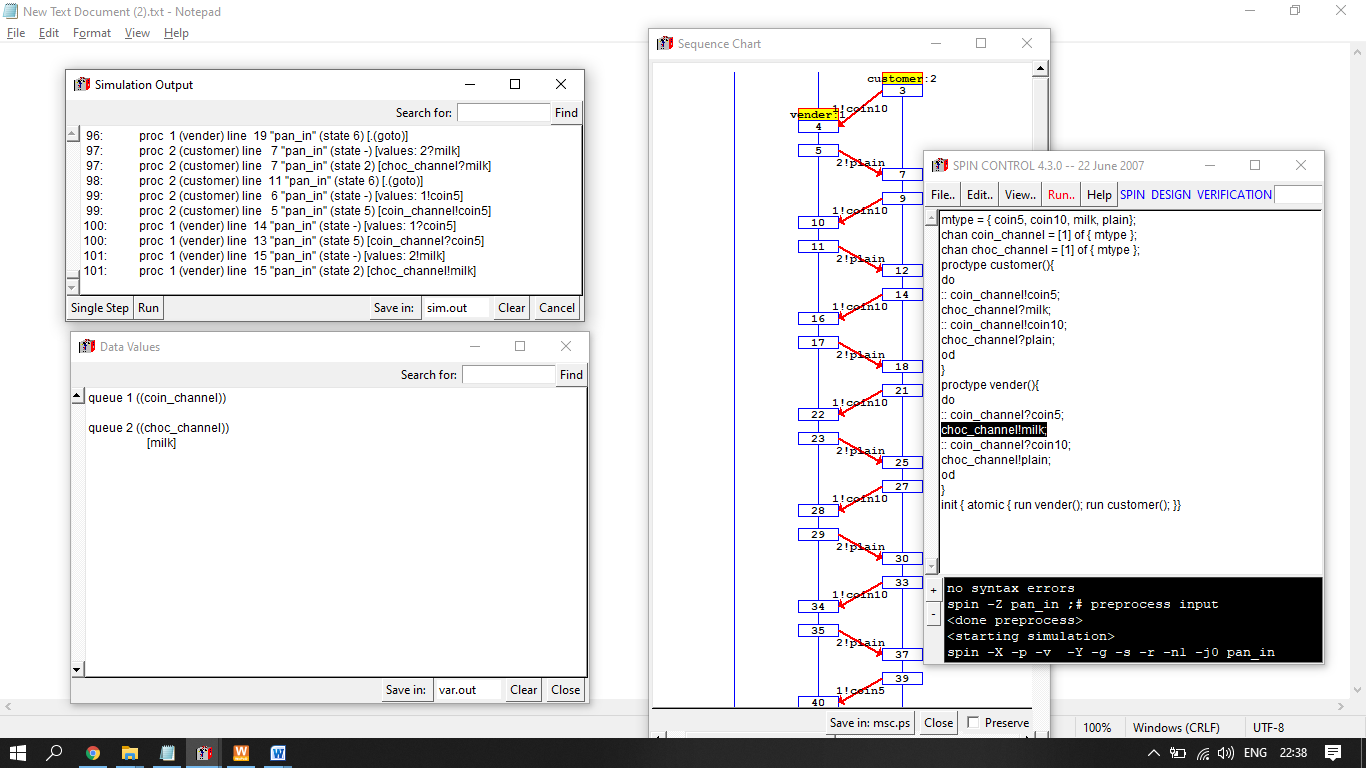
:: coin\_channel?coin10;

choc\_channel!plain;

od

}

init { atomic { run vender(); run customer(); }}



**6.**

mtype = { open, close }

type out=open, in=close;

byte water\_level = 25;

byte user\_water = 0; /\* user reservoir \*/

proctype sensors(){

do

:: atomic{ (water\_level <= 20) -> out=close; in=open; }

:: atomic{ (water\_level >= 30) -> out=open; in=close; }

od

}

proctype user(){

do

:: (user\_water > 0) -> user\_water=user\_water-1;

:: true -> skip;

od

}

proctype inlet(){

do ::(in == open) -> water\_level=water\_level+1; od

}

proctype outlet(){

do ::(out == open) -> atomic{ water\_level=water\_level-1;

user\_water=user\_water+1;}

od

}

proctype monitor(){ do :: assert( water\_level >= 20 &&

water\_level <= 30 ); od }

init{ atomic{ run monitor(); run sensors();

run user(); run inlet(); run outlet(); } }

**7.**

#define On 1

#define Off 0

mtype = {Red,Orange,Green}

byte s1\_status

byte s2\_status

bit sensor\_status

chan chan\_s1 = [0] of {byte}

chan chan\_s2 = [0] of {byte}

chan chan\_sensor = [0] of {bit};

proctype HighwaySignal(mtype status)

{

bit sense = Off;

s1\_status = status;

byte temp;

chan\_sensor ! sense

(sensor\_status == On)->s1\_status = Orange

(s1\_status == Orange)->s1\_status = Red

chan\_s1 ! s1\_status

chan\_s2 ? temp

(temp == Red)->s1\_status = Green

}

proctype FarmRoadSignal(mtype status)

{

s2\_status = status

byte temp;

bit sense;

chan\_s1 ? temp;

(sensor\_status == On && temp == Red)-> s2\_status = Green

sensor\_status = Off

(sense == Off) -> s2\_status = Red

chan\_s2 ! s2\_status

}

proctype Sensor()

{

do

:: chan\_sensor ? sensor\_status

:: (sensor\_status == Off) -> sensor\_status = On

od //chan\_sensor ? sensor\_status

}

init

{

atomic{run HighwaySignal(Green);run FarmRoadSignal(Red);run Sensor()}

}

**8.**

mtype = {msg, ack};

chan to\_sndr = [2] of {mtype, bit};

chan to\_rcvr = [2] of {mtype, bit};

proctype Sender(chan in, out)

{

bit sendval, recval;

do

:: out!msg(sendval) -> in?ack(recval);

if

:: (recval == sendval) ->

sendval = 1 - recval;

:: else -> skip

fi

od

}

proctype Receiver(chan in, out)

{

bit recval;

do

:: in!msg(recval) ->

out?ack(recval);

:: timeout ->

out?ack(recval);

od

}

init {

run Sender (to\_sndr, to\_rcvr);

run Receiver(to\_rcvr, to\_sndr);29. }

**9.**

chan pwr\_on = [0] of {int}

chan pwr\_off = [0] of {int}

chan fan\_on = [0] of {int}

chan fan\_off = [0] of {int}

int x1,x2,x3,x4,x5;

proctype Idle()

{

do

:: pwr\_on ! x1

// Transition from Idle to power through channel pwr\_on

pwr\_off ? x2

pwr\_off ? x5

od

}

proctype Power()

{

pwr\_off ! x2

// Transition from Power to Idle through channel pwr\_off

fan\_on ! x3

// Transition from Power to Fan through channel fan\_on

fan\_off ? x4

}

proctype Fan()

{

fan\_off ! x4

pwr\_off ! x5

}

init

{

x1=1

x2=2

x3=3

x4=4

x5=5

atomic{run Idle();run Power();run Fan()}

}