
TUGAS KULIAH

METODE FORMAL (EC7030)

Verifikasi Beberapa Algoritma penanganan Proses
untuk Penyelesaian Masalah Mutual Exclusion
dengan PROMELA dan SPIN

Muhammad Nasrun
232 02 017

Departemen Teknik Elektro
Fakultas Teknik Industri
Institut Teknologi Bandung



DAFTAR ISI

DAFTAR ISI	i
ABSTRAK	ii
I PENDAHULUAN	1
1.1 Ilustrasi Printer Daemon	1
1.2 Ilustrasi Aplikasi Tabungan	3
1.3 Pokok Bahasan	4
II TINJAUAN UMUM MUTUAL EXCLUSION	5
2.1 Mutual Exclusion	5
2.1.1 Deadlock	5
2.1.2 Startvation	6
2.2 Masalah Critical Section	6
2.3 Metode Penyelesaian Masalah <i>Mutual Exclusion</i>	7
2.3.1 Sinkronisasi	7
2.3.2 Semaphore	12
III SPESIFIKASI DAN RANCANGAN MODEL	14
3.1 Spesifikasi Model	14
3.2 Pemodelan	14
3.2.1 Algoritma 3 (Peterson)	14
3.2.2 Algoritma Dekker	14
3.2.3 Algoritma Bakery	15
3.2.4 Penyelesaian Mutual Exclusion Umum	15
3.2.5 Dijkstra Semaphore	17
3.2.6 Semaphore	17
IV SIMULASI DAN VERIFIKASI MODEL	19
4.1 Simulasi	19
4.1.1 Simulasi Algoritma 3 Peterson	20
4.1.2 Simulasi Algoritma Dekker	21
4.1.3 Simulasi Algoritma Bakery	24
4.1.4 Simulasi Mutual Exclusion Umum	26
4.1.5 Simulasi Algoritma Dijkstra Semaphore	28
4.1.6 Simulasi Semaphore	30
4.2 Verifikasi	32
Verifikasi Algoritma 3 Peterson	32
Verifikasi Algoritma Dekker	34
Verifikasi Algoritma Bakery	35
Verifikasi Mutual Exclusion Umum	37
Verifikasi Algoritma Dijkstra Semaphore	39
Verifikasi Semaphore	41
V KESIMPULAN	48
DAFTAR PUSTAKA	49

ABSTRAK

Mutual exclusion dapat mencegah proses kongkurensi (concurrency) dikerjakan bersamaan oleh resource (sumber daya). Sehingga dibutuhkan penanganan proses kongkurensi tersebut agar dapat dikerjakan oleh resource. Metode penyelesaian masalah mutual exclusion ini adalah mengatur proses yang akan memasuki critical section. Jika proses yang sedang mengerjakan critical section, maka proses lain tidak boleh memasuki critical section. Ada beberapa metode yang dapat menyelesaikan permasalahan ini. Prinsip kerja antara satu metode dengan metode lainnya didalam mengatur penanganan mutual exclusion adalah berbeda. Dalam bahasan ini akan membahas bagaimana kinerja beberapa metode tersebut dalam menangani masalah mutual exclusion jika terjadi proses kongkurensi. Cara meninjau metode ini adalah dengan cara memverifikasi kinerja setiap metode dengan menggunakan tools SPIN dan bahasa pemrograman Promela.

BAB 1

PENDAHULUAN

Perkembangan sistem komputer mendatang adalah menuju ke sistem multi-processing, multiprogramming, terdistribusi dan paralel yang mengharuskan adanya proses-proses yang berjalan bersama dalam waktu yang bersamaan. Hal demikian merupakan masalah yang perlu perhatian dari perancang sistem operasi. Kondisi dimana pada saat yang bersamaan terdapat lebih dari satu proses disebut dengan kongkurensi (*concurrency*) atau proses-proses yang kongkuren. Proses-proses yang mengalami kongkuren dapat berdiri sendiri (*independen*) atau dapat saling berinteraksi, sehingga membutuhkan sinkronisasi atau koordinasi proses yang baik. Untuk penanganan kongkuren, bahasa pemrograman saat ini telah memiliki mekanisme kongkurensi dimana dalam penerapannya perlu dukungan sistem operasi dimana bahasa berada.

Karena pentingnya dukungan sistem operasi dalam penanganan kongkurensi, maka kongkurensi merupakan landasan umum dari perancangan sistem operasi. Sehingga jika sistem operasi tidak dapat menangani kongkurensi, akan mengakibatkan masalah pada saat proses-proses yang bersamaan tersebut akan dieksekusi. Permasalahan yang memungkinkan terjadi adalah saat proses-proses yang berjalan bersama dalam waktu yang bersamaan (kongkurensi) menggunakan *resource* (sumber daya) yang sama dalam eksekusi prosesnya. Dimana ada beberapa *resource* dalam sistem komputer yang tidak dapat mengeksekusi proses yang banyak sekaligus dalam waktu yang bersamaan, prinsip ini dikenal dengan *mutual exclusion*. Kejadian seperti ini jika dipaksakan akan mengakibatkan ketidakstabilan sistem operasi dalam mengatur pemrosesan sistem dalam sistem komputer. Untuk lebih jelasnya dapat diilustrasikan dalam dua ilustrasi berikut [1] :

1. Ilustrasi eksekusi *daemon* printer.
2. Ilustrasi aplikasi tabungan

1.1 Ilustrasi Printer Daemon.

Daemon printer adalah proses penjadwalan dan pengendalian pencetakan berkas-berkas di printer sehingga seolah-olah printer dapat digunakan secara simultan oleh proses-proses. Daemon printer mempunyai ruang *disk* (disebut direktori *spooler*) untuk menyimpan berkas-berkas yang akan dicetak. Direktori *spooler* membagi *disk*

menjadi sejumlah slot. Slot-slot diisi berkas yang akan dicetak. Terdapat variable *in* menunjuk slot bebas di ruang *disk* yang kan dipakai menyimpan berkas yang ingin dijadwalkan untuk dicetak. Perhatikan program dibawah ini :

```

Program Give File to spooler;
Var
    in : integer;
    BerkasA, berkasB : file;

Procedure ProsesA;
Var
    next_free_slot : integer;
Begin
    next_free_slot:=in;
    store(berkasA, next_free_slot);
    in:=next_free_slot + 1;
End;

Procedure ProsesB;
Var
    next_free_slot : integer;
Begin
    next_free_slot:=in;
    store(berkasB, next_free_slot);
    in:=next_free_slot + 1;
End;

Begin
    in:=0;
    Repeat
        Parbegin
            ProsesA;
            ProsesB;
        Parend
    Forever
End.

```

Skenario yang Membuat Kacau

Misal proses A dan B ingin mencetak berkas, variable *in* bernilai 9. scenario berikutdapat terjadi :

Proses A	Proses B
<pre> {in=9} next_free_slot ? in {in=9} {penjadwal menjadwalkan B berjalan} {in=9} store berkasB to slot[next_free_slot] {berkasB disimpan di slot ke 9 menimpa berkas B} in ? next_free_slot + 1 {in=10} </pre>	<pre> {in=9} next_free_slot ? in store berkasB to slot[next_free_slot] {berkasB disimpan di slot ke 9} in ? next_free_slot + 1 {in=10} { penjadwal menjadwalkan A berjalan} </pre>
<p>{berkasA menimpa berkasB, sehingga B tak pernah mendapat cetaknya}</p>	

Skenario

Proses A membaca variable *in* bernilai 9. Belum sempat A menyelesaikan proses, penjadwal menjadwalkan proses B berjalan. Proses B yang juga ingin mencetak, membaca variable *in*, variable *in* masih berniali 9. Proses B dapat menyelesaikan proses. Proses B menyimpan berkas B dislot ke-9. Proses A dijadwalkan kembali dan juga menyimpan berkas A di slot ke 9. BerkasB ditima berkas A, B tak akan pernah memperoleh cetakan.

Pada contoh terdapat kondisi dimana dua proses atau lebih sedang membaca atau menulis data bersama (yaitu variable *in*) denga hasil akhir bergantung jalannya proses-proses. Hasil akhir tidak dapat diprediksi. Kondisi ini disebut kondisi pacu (*race condition*). Kondisi pacu harus dihilangkan agar hasil proses dapat diprediksi dan tidak bergantung jalannya proses-proses itu.

Kunci penghilangan kondisi pacu adalah harus dapat dicegah lebih dari satu proses membaca atau menulis data bersama pada saat bersamaan. Mutual exclusion adalah menjamin hanya satu proses yang sedang menggunakan variable atau berkas pada suatu saat. Proses-proses lain dilarang mengerjakan hal yang sama. Bagian program yang sedang mengakses memori atau sumber daya yang dipakai bersama disebut *critical section/region*.

Untu mengatasi kondisi pacu harus dijamin tidak boleh dua proses atau lebih memasuki *critical section* yang sama secara bersamaan. Kesuksesan proses-proses konkuren memerlukan pendefinisian *critical section* dan memaksakan *mutual exclusion* di antara proses-proses kongkuren yang sedang berjalan. Pemaksaan *mutual exclusion* merupakan landasan pemrosesn kongkuren.

1.2 Ilustrasi Aplikasi i Tabungan

Seluruh sistem yang melibatkan banyak proses mengakses satu sumber daya bersama selalu menimbulkan masalah *mutual exclusion*. Ilustrasi ini dapat diperlihatkan pada proses update data pada plikasi tabungan sebagai berikut :

Pada aplikasi tabungan, misalnya rekening A berisi Rp 1.000.000,- terdaftar di kantor cabang di Bandung. Pada suatu saat program aplikasi dikantor cabang di Jakarta melayani penyetoran Rp 3.000.000,- ke rekening A tersebut. Program palikasi membaca saldo akhir rekening A. Pada waktu bersamaan, dikantor

cabang di Bandung juga terjadi transaksi yaitu penyetoran Rp 5.000.000,- ke rekening A. Program aplikasi di Bandung membaca saldo masih Rp 1.000.000.

Beberapa scenario dapat terjadi bila mutual exclusion tidak dijamin, yaitu :

1. Program aplikasi di Bandung dilakukan secara cepat menulis ke rekening A sehingga dihasilkan RP 6.000.000. Setelah itu program aplikasi dikantor cabang Jakarta menimpa hasil tersebut dengan Rp. 4.000.000. Hasil akhir diperoleh adalah Rp. 4.000.000,- yang seharusnya RP. 10.000.000,-
2. Program Aplikasi di Jakarta dilakukan secara cepat menulis ke rekening A sehingga dihasilkan Rp. 4.000.000. Setelah itu program aplikasi di kantor cabang Bandung menimpa hasil tersebut dengan Rp 6.000.000. Hasil lebih baik dibanding scenario pertama, tetapi masih dibawah yang seharusnya yaitu Rp. 10.000.000.

Masalah seperti ini menjadi bahasan sistem operasi dan pembuat manajemen basis data.

1.3 Pokok Bahasan

Dari ilustrasi kedua contoh diatas dapat kita simpulkan bahwa dalam mengatasi proses konkurensi membutuhkan pendefinisian *critical section* dan memaksakan *mutual exclusion* diantara proses-proses tersebut yang sedang berjalan. Jika hal tersebut tidak dapat diselesaikan dengan prinsip *mutual exclusion* dapat memungkinkan terjadi *deadlock* dan *startvation* (pembahasan dalam bab 2). Karena pentingnya penerapan mutual exclusion dalam mengendalikan proses konkurensi dalam pemrosesannya, maka diperlukan bahasan untuk meninjau ke-efektif-an beberapa metode penjaminan mutual exclusion. Dimana peninjauan yang dilakukan adalah memverifikasi beberapa metode penjaminan mutual exclusion untuk penyelesaian pemrosesan proses konkurensi. Tools verifikasi yang digunakan adalah **Program SPIN (Simple Promela Interpreter)**, salah satu tools yang dapat bersifat model checker untuk menverifikasi beberapa sistem. Bahasa pemrograman yang digunakan adalah bahasa **PROMELA (Protocol/Process Meta Language)**. Bahasa promela ini menyerupai bahasa C (ditambah feature dari CSP).

BAB II

TINJAUAN UMUM MUTUAL EXCLUSION

2.1 Mutual Exclusion

Merupakan kondisi dimana terdapat sumber daya yang tidak dapat dipakai bersama pada waktu yang bersamaan (misalnya : printer, disk drive). Kondisi demikian disebut sumber daya kritis, dan bagian program yang menggunakan sumber daya kritis disebut *critical region / section*. Hanya satu program pada satu saat yang diijinkan masuk ke *critical region*. Pemrogram tidak dapat bergantung pada sistem operasi untuk memahami dan memaksakan batasan ini, karena maksud program tidak dapat diketahui oleh sistem operasi. Hanya saja, sistem operasi menyediakan layanan (*system call*) yang bertujuan untuk mencegah proses lain masuk ke *critical section* yang sedang digunakan proses tertentu. Pemrograman harus menspesifikasikan bagian-bagian *critical section*, sehingga sistem operasi akan menjaganya. Pemaksaan atau pelanggaran *mutual exclusion* menimbulkan :

- a. Deadlock
- b. Startvation

2.1.1 Deadlock.

Proses disebut deadlock jika proses menunggu suatu kejadian tertentu yang tak akan pernah terjadi. Sekumpulan proses berkondisi *deadlock* bila setiap proses yang ada dikumpulan itu menunggu suatu kejadian yang hanya dapat dilakukan proses lain yang juga berada dikumpulan itu. Proses menunggu kejadian yang tak akan pernah terjadi.

Deadlock terjadi ketika proses-proses mengakses secara eksklusif sumber daya. Semua *deadlock* yang terjadi melibatkan persaingan memperoleh sumber daya eksklusif oleh dua proses atau lebih.

Terjadi *trade-off* antara overhead mekanisme penyelesaian deadlock dan manfaat-manfaat yang diperoleh. Pada beberapa kasus, ongkos yang harus dibayar untuk membebaskan sistem dari *deadlock* adalah mahal. Pada kasus-kasu tertentu, seperti pada sistem pengendalian proses waktu nyata (*real-time process control system*), tak ada pilihan kecuali membayar semua kemahalan karena adanya *deadlock* akan mengakibatkan sistem kacau.

2.1.2 Startvation.

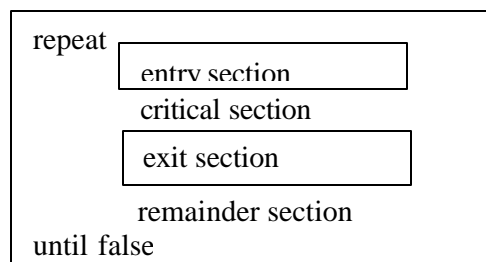
Proses dikatakan sebagai mengalami *startvation* bila proses-proses itu menunggu alokasi sumber daya sampai tak terhingga, sementara proses-proses lain dapat memperoleh alokasi sumber daya, *startvation* disebabkan bias pada kebijaksanaan atau strategi alokasi sumber daya. Kondisi ini harus dihindari karena tidak adil tapi dikehendaki penghindaran dilakukan seefisien mungkin. Sungguh merupakan persoalan yang sulit mempertemukan kriteria benar, adil dan efisien dalam satu strategi.

2.2 Masalah Critical Section.

Critical Section adalah suatu bagian yang berisi sejumlah variabel yang akan di-share (dipengaruhi dan mempengaruhi) proses yang lain.

Secara umum, penyelesaian *critical section* harus memenuhi 3 syarat [2]:

1. *Mutual exclusion*. Jika suatu proses sedang mengerjakan *critical section*, maka tidak boleh ada proses lain yang masuk (mengerjakan) *critical section* tersebut.
2. *Progress*. Jika tidak ada suatu proses yang mengerjakan *critical section* dan ada beberapa proses yang akan masuk *critical section*, maka hanya proses-proses yang sedang berada pada *entry-section* saja yang boleh berkompetisi untuk mengerjakan *critical action*.
3. *Bounded waiting*. Besarnya waktu tunggu dari suatu proses yang akan memasuki *critical section* sejak proses itu meminta ijin untuk mengerjakan *critical section*, hingga permintaan itu dipenuhi.



Gambar 2.1 Struktur untuk Proses Pi

Untuk setiap proses yang akan masuk *critical section* harus meminta ijin terlebih dahulu, dan proses yang mendapat ijinlah yang akan masuk ke *critical section*. Gambar diatas menunjukkan struktur untuk proses Pi. *Entry-section* adalah daerah tempat proses

menunggu untuk memasuki *critical section*. Sedangkan *exit-section* adalah daerah dimana suatu proses baru saja keluar dari *critical section*.

Menurut [3] [4] kriteria penyelesaian *Mutual Exclusion* harus memenuhi kriteria-kriteria berikut :

1. *Mutual exclusion* harus dijamin.
Hanya satu proses pada satu saat yang diijinkan masuk *critical section*. Proses-proses lain dilarang masuk *critical section* yang sama pada saat telah terdapat proses masuk di *critical section* itu.
2. Proses yang berada di *noncritical section*, dilarang mem-blocked proses-proses lain yang ingin masuk *critical section*.
3. Harus dijamin proses yang ingin masuk *critical section* tidak menunggu selama waktu yang tak hingga atau tak boleh terdapat *deadlock* atau *starvation*.
4. Ketika tidak ada proses pada *critical section* maka proses yang ingin masuk *critical section* harus diijinkan masuk tanpa waktu tunda.
5. Tidak ada asumsi mengenai kecepatan relatif proses atau jumlah proses yang ada.

Kriteria 1 merupakan pokok yang harus dipenuhi. Metode yang melanggar kriteria sama sekali tidak dapat digunakan. Pelanggaran kriteria-kriteria lain berarti metode masih dapat digunakan pada situasi tertentu tapi harus dilakukan secara hati-hati.

Dari [2][3][4] dapat disimpulkan teknik penyelesaian *mutual exclusion* hampir sama secara keseluruhan.

2.3 Metode Penyelesaian Masalah *Mutual Exclusion*.

Ada beberapa metode penyelesaian masalah *mutual exclusion*, antara lain :

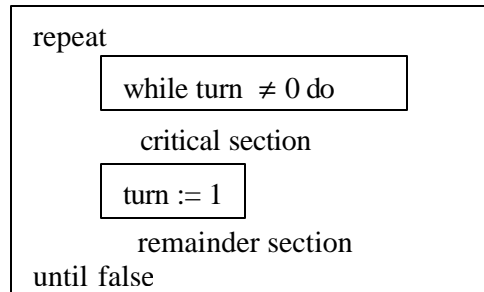
1. Sinkronisasi.
2. Semaphore.

2.3.1 Sinkronisasi

Pada bagian ini akan dibatasi pada aplikasi ke 2 proses yaitu P_i dan P_j , atau P_0 dan P_1 . Secara umum, jika ada proses P_i maka akan digunakan proses P_j sebagai proses lainnya, dengan $j=1-i$. Ada beberapa algoritma penyelesaian *mutual exclusion* dengan menggunakan sinkronisasi, yakni:

Algoritma 1.

Kedua proses akan berbagi suatu variabel bertipe integer yaitu *turn* yang diinisialisasikan dengan 0 (atau 1). Jika $turn=0$, maka proses P_0 diijinkan untuk mengeksekusi *critical section*. Struktur P_0 terlihat pada gambar dibawah ini :



Gambar 2.2 Struktur untuk Proses P_0 di Algoritma 1.

Solusi ini sudah menunjukkan adanya *mutual exclusion*, karena pada suatu saat hanya ada satu proses yang masuk *critical section*. Namun belum menunjukkan adanya progress dan *bounded waiting*. Sebagai contoh :

- Jika P_0 meninggalkan *critical section*, maka nilai $turn=1$ yang berarti bahwa P_1 siap untuk masuk ke *critical section*;
- P_1 selesai menggunakan *critical section* dengan cepat maka baik P_1 maupun P_0 beradapada *remainder section* dengan nilai $turn=0$.
- P_0 kembali menggunakan *critical section* dengan cepat dan segera masuk ke *remainder section* dengan nilai $turn=1$.
- P_0 hendak kembali menggunakan *critical section* namun nilai $turn=1$.

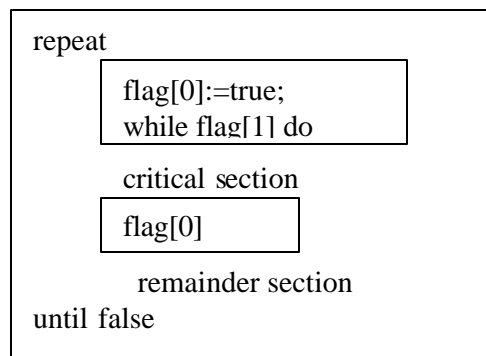
Terlihat bahwa P_1 yang berada pada *remainder section* memblok P_0 sehingga tidak dapat memasuki *critical section*. Hal ini menentang progress, yaitu proses diblok oleh proses yang tidak berada di *critical section*.

Algoritma 2.

Masalah pada algoritma-1 tidak memberikan cukup informasi mengenai status proses. Hanya mempertimbangkan yang masuk *critical section*. Untuk mengatasi masalah ini variabel “*turn*” diganti dengan :

```
var flag : array [0 ..1] of boolean;
```

Semua elemen array diinisialisasikan dengan false. Jika flag[0] bernilai true, maka nilai itu akan mengindikasikan bahwa P₀ siap memasuki *critical section*. Struktur P₀ terlihat pada gambar dibawah ini.



Gambar 2.3 Struktur untuk Proses P₀ di Algoritma 2

Pada algoritma ini, proses P₀ pertama kali menetapkan flag[i] = true, nilai ini mengindikasikan bahwa P₀ siap memasuki *critical section*. Kemudian P₀ mengecek untuk menyakinkan P₁ tidak akan memasuki *critical section*. Jika P₁ juga telah memasuki *critical section*, maka P₀ harus menunggu sampai P₁ tidak membutuhkan *critical section* lagi (sampai flag[1] =false). Secepatnya P₀ memasuki *critical section*. Pada *exit section*, P₀ akan mengeset flag[0] menjadi false, hal ini mengijinkan proses lain (jika sedang menunggu) untuk memasuki *critical section*.

Pada solusi ini, kondisi *mutual exclusion* telah dipenuhi. Namun progress belum juga dipenuhi. Untuk menggambarkan hal tersebut dapat dilihat :

T₀ P₀ menetapkan flag[0] = true;

T₁ P₁ menetapkan flag[1] = true;

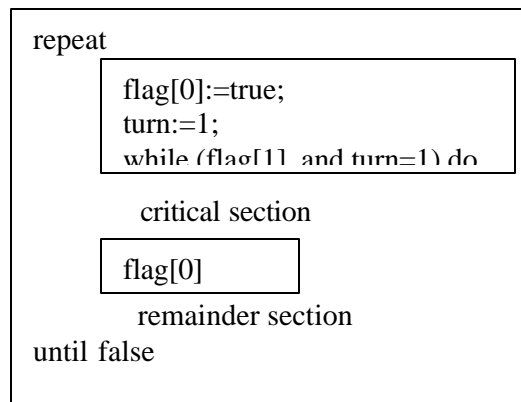
Sekarang P₀ dan P₁ bersama-sama ada di *statement while*.

Algoritma –3 (Peterson)

Algoritma ketiga ini diperkenalkan oleh Peterson. Pada dasarnya, algoritma ini merupakan kombinasi antara algoritma –1 dan algoritma-2. Proses ini membutuhkan 2 variabel, yaitu :

```
Var      Flag   :array [0 .. 1] of boolean;
        Turn    : 0 .. 1;
```

Nilai awal $\text{flag}[0] = \text{flag} = \text{false}$, dan nilai turn (0 atau 1). Struktur proses P_1 seperti terlihat pada gambar dibawah ini :



Gambar 2.4 Struktur untuk Proses P_0 di Algoritma 3.

Untuk masuk ke *critical section*, proses P_0 mengeset $\text{flag}[0] = \text{true}$, dan melihat apakah ada proses lain yang mencoba masuk *critical section* ($\text{turn}=1$).

Algoritma Bakery

Critical section untuk n proses:

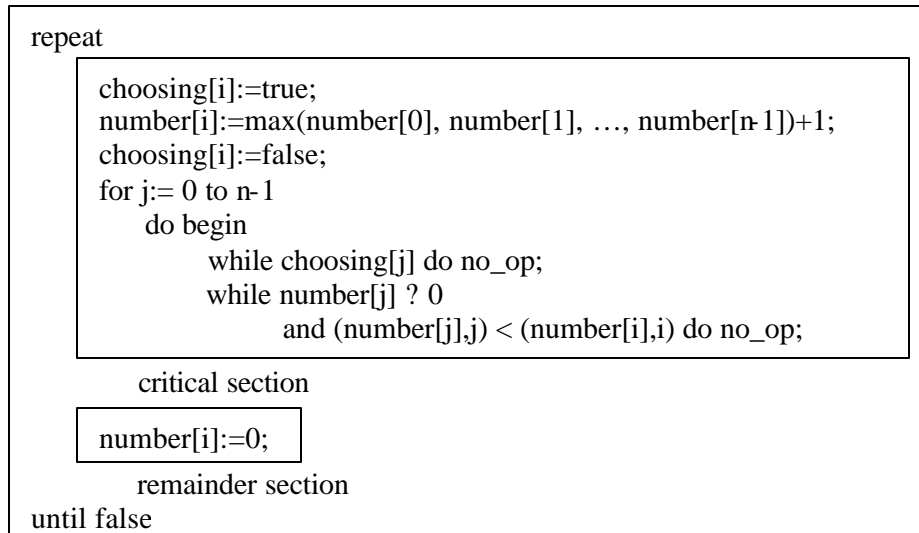
1. Sebelum masuk ke *critical section*. Proses mendapatkan nomor. Yang memiliki nomor paling kecil lebih dahulu memasuki *critical section*.
2. Jika proses P_i dan P_j menerima nomor, jika $i < j$, maka P_i lebih dahulu masuk *critical section*, sebaliknya P_j yang lebih dahulu masuk *critical section*.
3. skema penomoran selalu membangkitkan angka yang meningkat, seperti 1,2,3,4,5,6, ...

4. shared data

```
var choosing : array[0..n-1] of boolean;
```

```
number : array[0..n-1] of integer;
```

Struktur data diinisialisasikan dengan *false* dengan nilai 0. Struktur proses P_i seperti terlihat pada gambar dibawah ini :



Gambar 2.5 Struktur untuk Proses P_0 di Algoritma 3.

Algoritma Dekker

Algoritma ini diperkenalkan oleh Dekker, seorang matematikawan dari Belanda. Algoritma ini memiliki ciri-ciri khusus sebagai berikut:

- Tidak memerlukan instruksi-instruksi per angkat keras khusus.
- Proses yang beroperasi di *remainder section* tidak dapat mencegah proses lain yang ingin masuk *critical section*.
- Proses yang ingin masuk *critical section* akan segera memasuki kawasan tersebut jika dimungkinkan.

2.3.2 Semaphore

Semaphore adalah salah satu cara menangani *critical section*, yang dikemukakan oleh *Dijkstra*. Prinsip *semaphore* sebagai berikut:

Dua proses atau lebih dapat bekerja sama dengan menggunakan penanda-penanda sederhana. Proses dipaksa berhenti sampai proses memperoleh penanda tertentu. Sembarang kebutuhan koordinasi kompleks dapat dipenuhi dengan penanda yang sesuai kebutuhannya. Variabel khusus untuk penandan ini disebut *semaphore*.

Semaphore mempunyai dua property, yaitu :

1. *Semaphore* dapat diinisialisasi dengan nilai nonnegative.
2. Terdapat dua operasi terhadap *semaphore* yaitu *Down* dan *Up*. Nama asli yang disampaikan *Dijkstra* adalah operasi P dan V.

Semaphore S merupakan variabel bertipe integer yang diakses dengan 2 standar operasi atomic, yaitu *wait* dan *signal*. Operasi-operasi ini diwakili dengan P (*wait*) dan V (*signal*) sebagai berikut:

```
wait(S)      :   while S ≤ 0 do no_op;
                S:=S - 1;
signal(S)    :   S:=S+1;
```

Misalkan ada 2 proses yang sedang berjalan secara konkuren, yaitu P₁ dengan pernyataan S₁ dan P₂ dengan pernyataan S₂. Andaikan kita mengharapkan S₂ baru akan dijalankan hanya setelah S₁ selesai. Hal ini dapat dilakukan dengan menggunakan bantuan *semaphore synch* (dengan nilai awal =0) yang akan di-*share* oleh kedua proses.

```
Untuk Proses P1 :
                S1;
                signal(synch);
Untuk proses P2 :
                wait(synch);
                S2;
```

Karena nilai awal untuk *synch* adalah nol, maka P₂ akan mengeksekusi S₂ hanya setelah P₁ mengerjakan *signal (synch)* setelah S₁.

Salah satu kerugian dari penggunaan semaphore diatas adalah adanya busy waiting. Apabila suatu proses menempati critical section, dan ada proses lain yang ingin masuk critical section, maka akan terjadi iterasi secara terus-menerus pada entry section. Hal ini akan menimbulkan masalah pada sistem yang menggunakan konsep *multiprogramming*.

Untuk menghindari busy waiting, dilakukan modifikasi pada operasi *wait* dan *signal*. Jika suatu proses sedang mengeksekusi operasi *wait*, maka nilai semaphore menjadi tidak positif. Pada saat ini proses akan memblok dirinya (*block*) dan ditempatkan pada *waiting queue*.

Proses yang sedang diblok akan menunggu hingga *semaphore S* direstart, yaitu pada saat beberapa proses yang lain mengeksekusi operasi *signal*. Suatu proses akan direstart dengan operasi *wakeup*, yang akan mengubah proses dari keadaan *waiting* ke *ready*.

Untuk mengimplmentasikan hal ini, *semaphore* dirancang dalam bentuk *record* :

```
type semaphore=      Record
                      value : integer;
                      L: list of process;
                      end;
Var s: semaphore
```

Operasi-operasi pda semaphore;

```
wait(S)      S.value :=S.value-1;
              if S.value < 0 then
                Begin
                  Tambahkan proses ini ke S.L.
                  block;
                end;
```

BAB III

SPESIFIKASI DAN RANCANGAN MODEL

3.1 Spesifikasi Model

Spesifikasi yang dibuat dalam membangun model adalah sesuai dengan prinsip kerja algoritma pada setiap penyelesaian masalah mutual exclusion pada bahasan bab 2.

3.2 Pemodelan

Model dari beberapa metode penyelesaian masalah *mutual exclusion* (dibahas dalam bab 2) dibangun dalam bahasa PROMELA) sebagai berikut :

3.2.1 Algoritma 3 (Peterson)

Model untuk algoritma 3 (Peterson) adalah sebagai berikut :

```
#define true 1
#define false 0
bool flag[2];
bool turn;
proctype user(bool i)
{
    flag[i] = true;
    turn = i;
    (flag[1-i] == false || turn == 1-i);
crit: skip; /* critical section */
    flag[i] = false
}
init { atomic { run user(0); run user(1) } }
```

3.2.2 Algoritma Dekker

Model untuk algoritma Dekker adalah sebagai berikut :

```
Bit x, y; /* signal masuk/keluar dari section */
byte mutex; /* # proses yang masuk critical section. */
byte turn,A_TURN,B_TURN; /* giliran siapa? */

active proctype A() {
x = 1;
turn = B_TURN;
(y == 0 || turn == A_TURN);
mutex++;
mutex--;
x = 0;
}

active proctype B() {
y = 1;
turn = A_TURN;
(x == 0 || turn == B_TURN);
```

```

mutex++;
mutex--;
y = 0;
}

active proctype monitor() {
assert(mutex != 2);
}

```

3.2.3 Algoritma Bakery

Model untuk algoritma Bakery adalah sebagai berikut :

```

Byte turn[2]; /* giliran siapa? */
byte mutex; /* # prose yang masuk ke critical section */

proctype P(bit i) {
do
:: turn[i] = 1;
   turn[i] = turn[1-i] + 1;
   (turn[1-i] == 0) || (turn[i] < turn[1-i]);
   mutex++;
   mutex--;
   turn[i] = 0;
od
}

proctype monitor() { assert(mutex != 2); }

init { atomic {run P(0); run P(1); run monitor()}}

```

3.2.4 Penyelesaian Mutual Exclusion Umum

Model untuk penyelesaian mutual exclusion secara umum dengan meng-share penggunaan resource adalah sebagai berikut :

```

bool Fail = false;
int share = 6;
int x = 0;
int y = 0;
bool z = true;

chan q_0_1 = [0] of {bit};
chan q_0_2 = [0] of {bit};

proctype A() {
bool dapat_dishare = true;
do
::atomic{
   z ->
   x = 10;
};
::atomic{
   dapat_dishare ->

```

```

    q_0_1?0;
    share = share+1;
    dapat_dishare = false;
    q_0_1?1;
};
::atomic{
    !dapat_dishare ->
    q_0_2?0;
    dapat_dishare = true;
    q_0_2?1;
};
od;
}

proctype B() {
bool dapat_dishare = false;
do
    ::atomic{
        z ->
        y = 11;
    };
    ::atomic{
        dapat_dishare ->
        q_0_2!0;
        share = share-1;
        dapat_dishare = false;
        q_0_2!1;
    };
    ::atomic{
        !dapat_dishare ->
        q_0_1!0;
        dapat_dishare = true;
        q_0_1!1;
    };
od;
}

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

```

3.2.5 Dijkstra Semaphore

Model untuk algoritma *Dijkstra semaphore* dengan menggunakan *rendevous* adalah sebagai berikut :

```

#define p 0
#define v 1
chan sema = [0] of { bit };

proctype dijkstra()
{
    byte count = 1;
    do
        :: (count == 1) -> sema!p; count = 0

```

```

    :: (count == 0) -> sema? v; count = 1
  od
}

proctype user()
{
  do
    :: sema? p;
      /* critical section */
      sema!v;
      /* non-critical section */
  od
}
init
{
  run dijkstra(); run user();
  run user(); run user()
}

```

3.2.6 Semaphore

Model semaphore secara umum untuk penyelesaian mutual exclusion adalah sebagai berikut :

```

mtype = { idle, masuk, critical, keluar };
#define state mtype

bool Fail = false;
bool semaphore = false;

proctype USER_0() {
  state user_state = idle;
  do
    ::atomic{
      user_state==idle ->
      if
        ::user_state = idle;
        ::user_state = masuk;
      fi;
    };
    ::atomic{
      ((user_state==masuk)&&!semaphore) ->
      user_state = critical;
      semaphore = true;
    };
    ::atomic{
      ((user_state==masuk)&&semaphore) ->
      user_state = masuk;
    };
    ::atomic{
      user_state==critical ->
      if
        ::user_state = critical;
        ::user_state = keluar;
      fi;
    };
    ::atomic{
      user_state==keluar ->

```

```

    user_state = idle;
    semaphore = false;
};
od;
}

proctype USER_1() {
state user_state = idle;
do
  ::atomic{
    user_state==idle ->
    if
      ::user_state = idle;
      ::user_state = masuk;
    fi;

  };
  ::atomic{
    ((user_state==masuk)&&!semaphore) ->
    user_state = critical;
    semaphore = true;
  };
  ::atomic{
    ((user_state==masuk)&&(semaphore)) ->
    user_state = masuk;
  };
  ::atomic{
    user_state==critical ->
    if
      ::user_state = critical;
      ::user_state = keluar;
    fi;

  };
  ::atomic{
    user_state==keluar ->
    user_state = idle;
    semaphore = false;
  };
od;
}

init {
  atomic{
    run USER_0();
    run USER_1();
  };
}

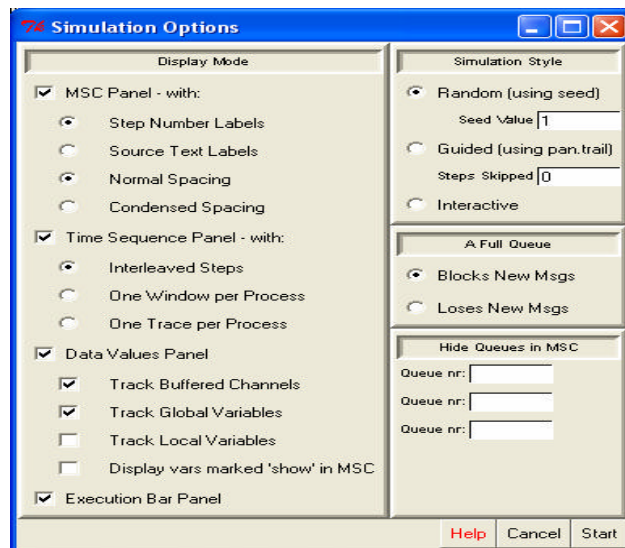
```

BAB IV SIMULASI DAN VERIFIKASI MODEL

4.1 Simulasi

Pada bagian ini dilakukan proses simulasi pada setiap model yang dibangun untuk melihat perilaku dari model dengan menggunakan SPIN.

Dengan menggunakan XSPIN, parameter simulasi dapat diatur dengan window seperti terlihat pada gambar 7 dibawah ini.



Gambar 4.1. Window parameter simulasi

Pada window diatas ada beberapa hal yang dapat diatur berkaitan dengan simulasi yang akan dilakukan. Yang pertama adalah panel apa saja yang akan ditampilkan selain panel standar *Simulation Output*. Panel *Message Sequence Chart* (MSC) akan menampilkan representasi grafis dari interaksi (pertukaran *message*) antar proses. Gambar yang muncul secara default dapat disimpan dalam file *msc.ps*. Panel *Time Sequence* (TS) menghasilkan representasi tekstual dari informasi yang serupa dengan yang muncul pada MSC. Tampilan dari TS ini dapat disimpan secara default ke file *seq.out*. Panel *Data Value* (DV) menghasilkan kondisi variabel-variabel lokal, global, maupun buffer selama simulasi berlangsung. Seluruh kondisi variable dapat disimpan secara default ke file *var.out*. Panel *Execution Bar* (EB) menampilkan

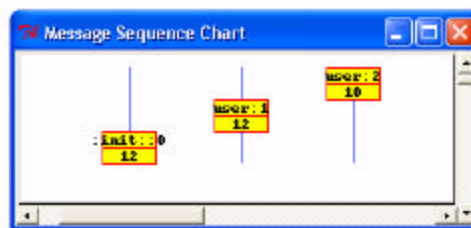
informasi tentang berapa banyak (persentase) *step* yang dieksekusi oleh tiap proses. Tampilan grafis EB dapat disimpan secara default ke file panbar.ps.

Simulasi dari setiap model penyelesaian masalah mutual exclusion yang dibangun sebagai berikut:

4.1.1 Simulasi Algoritma 3 (Peterson)

Simulasi algoritma Peterson dengan menggunakan tools SPIN sebagai berikut :

- a. Snapshot panel Message Sequence Chart.



Gambar 4.2 Tampilan Akhir MSC

- b. Snapshot panel Simulation output :

```
0: proc -(root) creates proc 0 (init)
1: proc 0 (init) creates proc 1 (user)
1: proc 0 (init) line 12 "pan_in" (state 3) [[run user(0)]]
```

Gambar 4.3 Awal Simulasi

```
0: proc -(root) creates proc 0 (init)
1: proc 0 (init) creates proc 1 (user)
1: proc 0 (init) line 12 "pan_in" (state 3) [[run user(0)]]
2: proc 0 (init) creates proc 2 (user)
2: proc 0 (init) line 12 "pan_in" (state 2) [[run user(1)]]
3: proc 2 (user) line 6 "pan_in" (state 1) [flag[] = 1]
4: proc 1 (user) line 6 "pan_in" (state 1) [flag[] = 1]
5: proc 2 (user) line 7 "pan_in" (state 2) [turn = i]
6: proc 1 (user) line 7 "pan_in" (state 2) [turn = i]
7: proc 2 (user) line 8 "pan_in" (state 3) [flag[0] = 0][turn == (1-i)]
8: proc 2 (user) line 9 "pan_in" (state 4) [1]
9: proc 2 (user) line 10 "pan_in" (state 5) [flag[] = 0]
10: proc 1 (user) line 8 "pan_in" (state 3) [flag[0] = 0][turn == (1-i)]
10: proc 2 (user) terminates
11: proc 1 (user) line 9 "pan_in" (state 4) [1]
12: proc 1 (user) line 10 "pan_in" (state 5) [flag[] = 0]
12: proc 1 (user) terminates
12: proc 0 (init) terminates
3 processes created
```

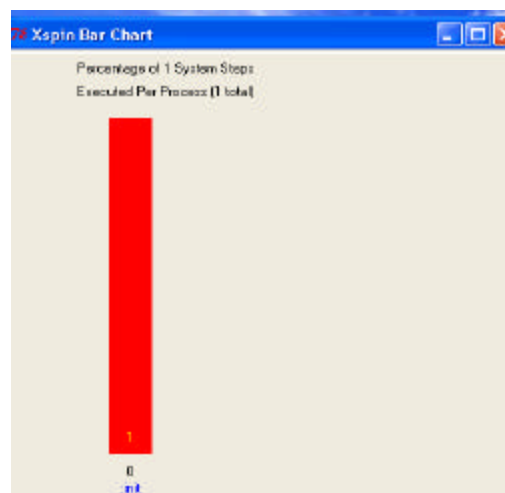
Gambar 4.4 Akhir Simulasi

c. Snapshot panel Data value :

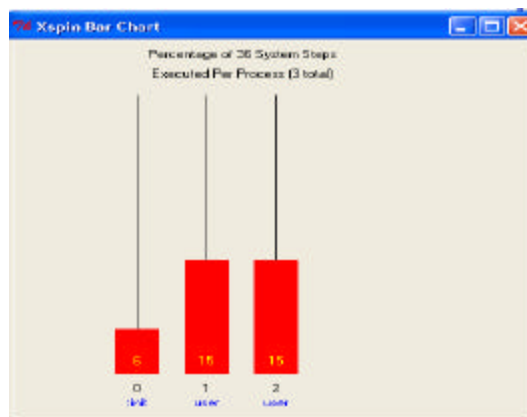


Gambar 4.5 Data Value

d. Snapshot panel execution bar chart



Gambar 4.6 Awal execution bar chart



Gambar 4.7 Akhir execution bar chart

Dari simulasi dapat dilihat bahwa 2 proses A dan B dapat saling bergantian memasuki critical section. Prinsip masuk ke *critical section*, proses P_0 mengeset $flag[0] = true$, dan melihat apakah ada proses lain yang mencoba masuk

critical section (turn=1). Jika tidak ada maka P_0 masuk ke *critical section*.

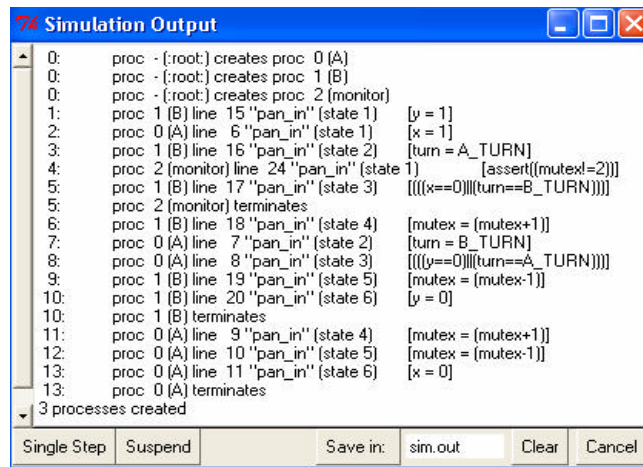
4.1.2 Simulasi Algoritma Dekker

Simulasi algoritma Dekker dengan menggunakan tools SPIN sebagai berikut :

a. Snapshot panel Simulation output :



Gambar 4.8 Awal Simulasi

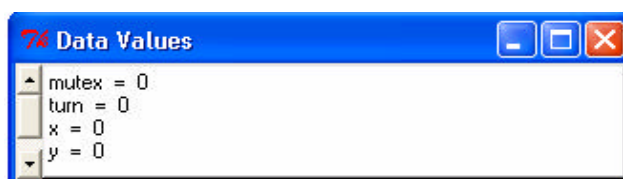


Gambar 4.9 Akhir Simulasi

b. Snapshot panel Data value :

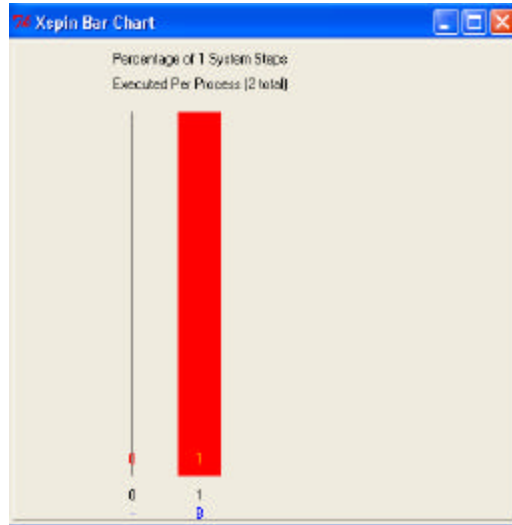


Gambar 4.10 Awal Data Value

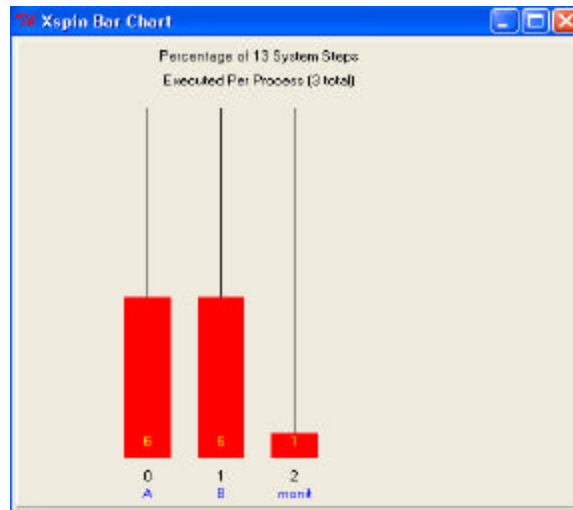


Gambar 4.11 Akhir Data Value

d. Snapshot panel execution bar chart



Gambar 4.12 Awal execution bar chart



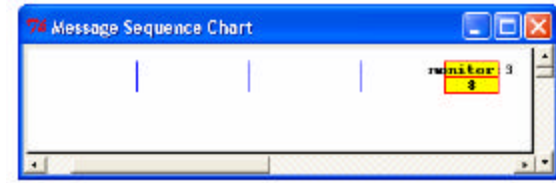
Gambar 4.13 Akhir execution bar chart

Dari simulasi dapat dilihat bahwa 2 proses A dan B dapat saling bergantian memasuki critical section. Tetapi dari simulasi terlihat jika proses lain ingin masuk ke critical section akan langsung masuk jika memungkinkan, tanpa melalui penanda;

4.1.3 Simulasi Algoritma Bakery

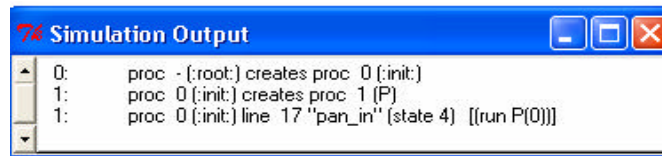
Simulasi algoritma 1 dengan menggunakan tools SPIN sebagai berikut :

a. Snapshot panel Message Sequence Chart.

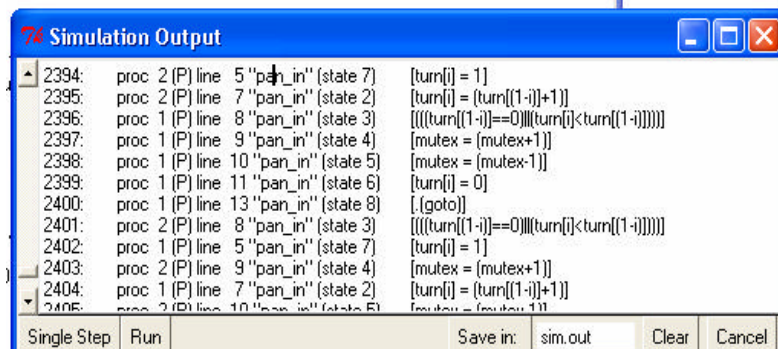


Gambar 4.14 Tampilan Akhir MSC

b. Snapshot panel Simulation output :



Gambar 4.15 Awal Simulasi



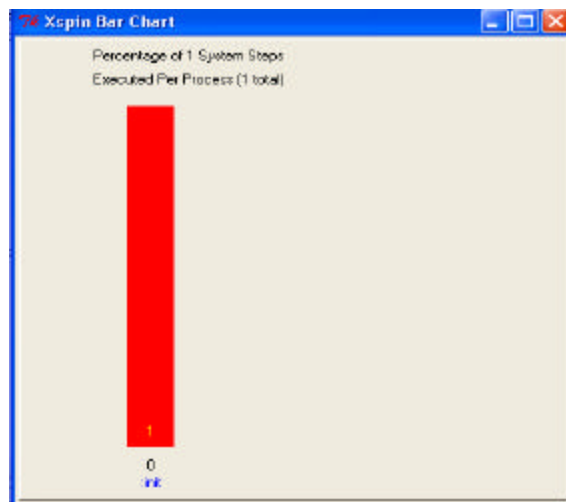
Gambar 4.16 Sebagian Simulasi

c. Snapshot panel Data value :

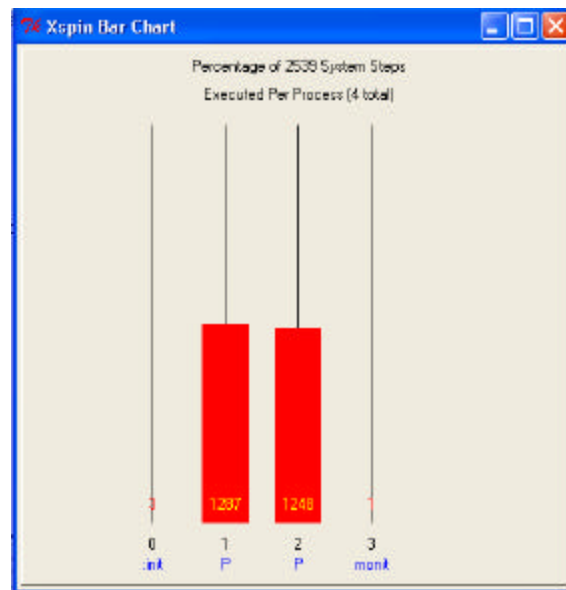


Gambar 4.17 Akhir Data Value

d. Snapshot panel execution bar chart



Gambar 4.18 Awal execution bar chart



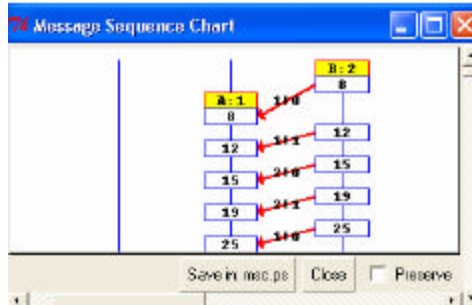
Gambar 4.18 Akhir execution bar chart

Dari simulasi dapat dilihat bahwa proses-proses dapat saling bergantian memasuki *critical section* melihat nomor proses tersebut, jika nomor proses tersebut kecil maka lebih dahulu masuk *critical section* dibanding proses yang nomornya lebih besar.

4.1.4 Simulasi Mutual Exclusion Umum

Simulasi algoritma 1 dengan menggunakan tools SPIN sebagai berikut :

a. Snapshot panel Message Sequence Chart.



Gambar 4.19 Tampilan Akhir MSC

b. Snapshot panel Simulation output :

```
0: proc -(root) creates proc 0 (init)
spin warning: 'par_in', global, 'b' Fail variable is never used
spin warning: 'par_in', global, 'i' variable is never used
spin warning: 'par_in', global, 'j' variable is never used
1: proc 0 (init) creates proc 1 (A)
1: proc 0 (init) line 60 'par_in' (state 3) [run A()]
```

Gambar 4.20 Awal Simulasi

```
740: proc 2 (B) line 46 'par_in' (state 6) [share = (share-1)]
741: proc 2 (B) line 47 'par_in' (state 7) [dapat_dishare = 0]
742: proc 2 (B) line 48 'par_in' (state 8) [q_0_271]
742: proc 1 (A) line 31 'par_in' (state 13) [q_0_271]
742: proc 2 (B) line 48 'par_in' (state -1) [values: 271]
742: proc 1 (A) line 31 'par_in' (state -1) [values: 271]
743: proc 1 (A) line 15 'par_in' (state 15) [i]
744: proc 1 (A) line 18 'par_in' (state 21) [k = 10]
745: proc 2 (B) line 39 'par_in' (state 15) [][dapat_dishare]]
746: proc 1 (A) line 34 'par_in' (state 16) [.[goto]]
```

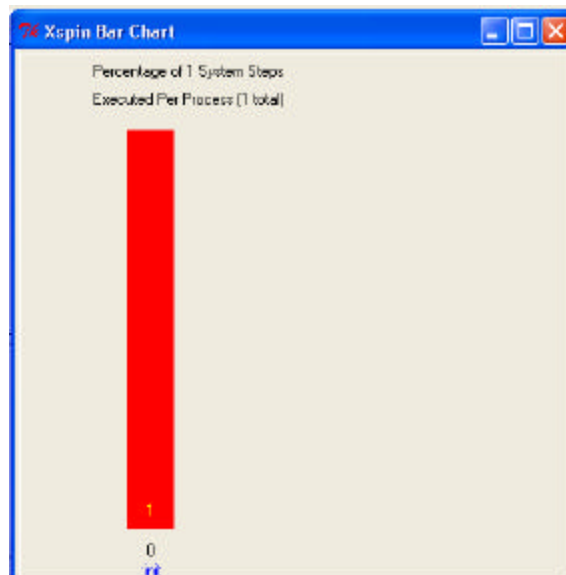
Gambar 4.21 Sebagian Simulasi

c. Snapshot panel Data value :

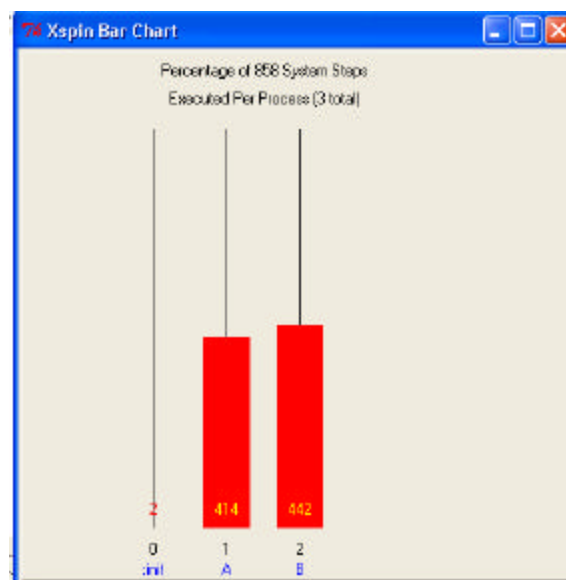
```
A[1]dapat_dishare = 1
B[2]dapat_dishare = 0
share = 6
k = 10
j = 11
```

Gambar 4.22 Akhir Data Value

d. Snapshot panel execution bar chart



Gambar 4.23 Awal execution bar chart



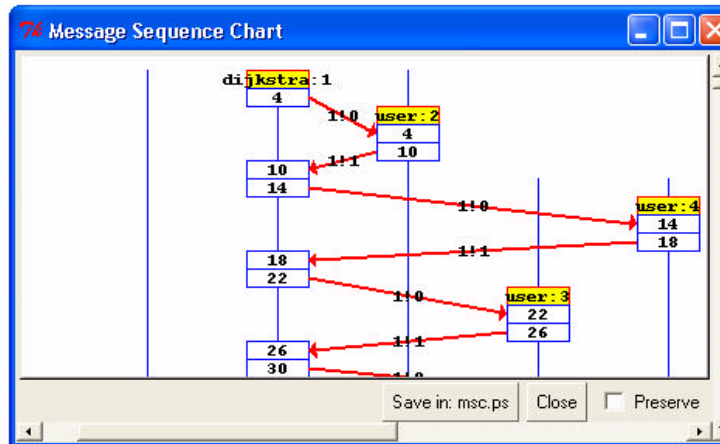
Gambar 4.24 Akhir execution bar chart

Dari simulasi dapat dilihat bahwa 2 proses A dan B dapat saling bergantian memasuki critical section. Pengaturan yang dilakukan adalah dengan saling meng-share resource yang dimiliki.

4.1.5 Simulasi Algoritma Dijkstra Semaphore.

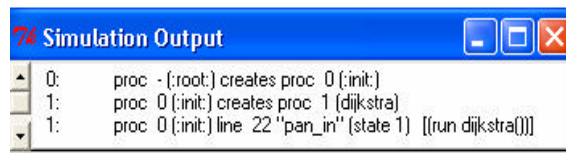
Simulasi algoritma Dijkstra semaphore dengan menggunakan tools SPIN sebagai berikut :

a. Snapshot panel Message Sequence Chart.

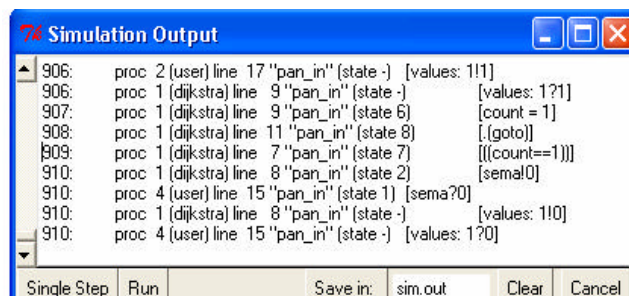


Gambar 4.25 Tampilan Akhir MSC

b. Snapshot panel Simulation output :



Gambar 4.26 Awal Simulasi



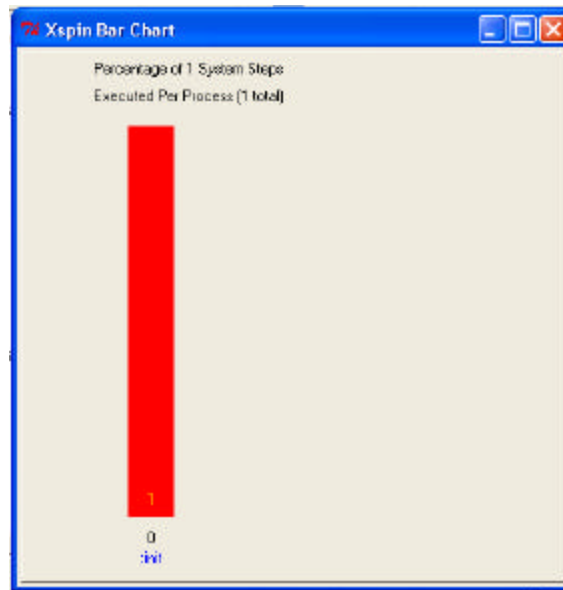
Gambar 4.27 Sebagian Simulasi

c. Snapshot panel Data value :

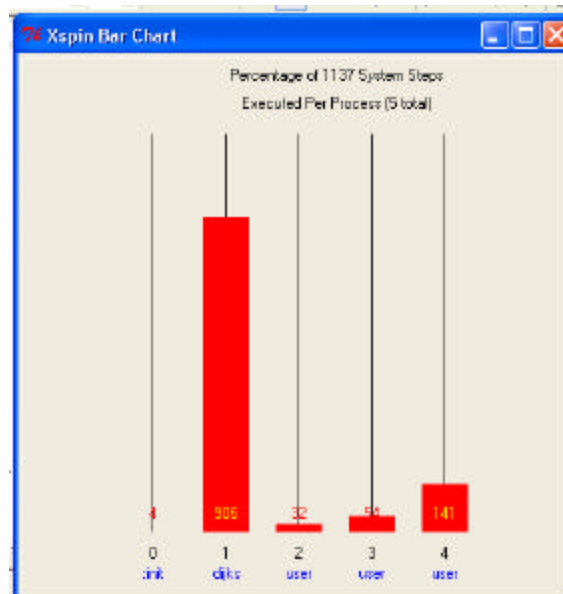


Gambar 4.28 Akhir Data Value

d. Snapshot panel execution bar chart



Gambar 4.29 Awal execution bar chart



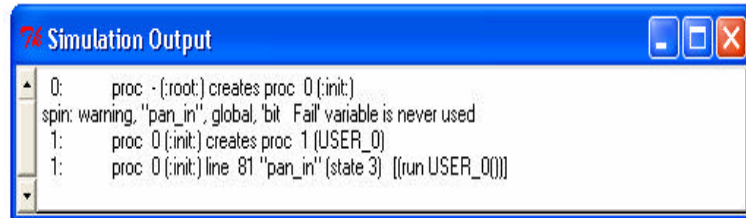
Gambar 4.30 Sebagian execution bar chart

Dari simulasi dapat dilihat bahwa proses-proses saling bergantian memasuki critical section dengan cara menggunakan penanda *signal* dan *wait*.

4.1.6 Simulasi Semaphore.

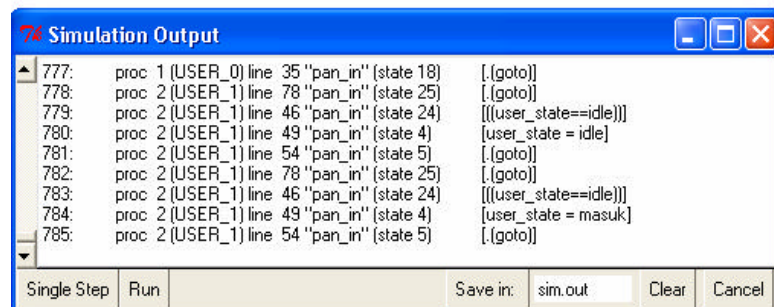
Simulasi algoritma semaphore dengan menggunakan tools SPIN sebagai berikut :

a. Snapshot panel Simulation output :



```
74 Simulation Output
0:   proc - (:root:) creates proc 0 (:init)
spin: warning, "pan_in", global, 'bit' Fail variable is never used
1:   proc 0 (:init) creates proc 1 (USER_0)
1:   proc 0 (:init) line 81 "pan_in" (state 3) [[run USER_0()]]
```

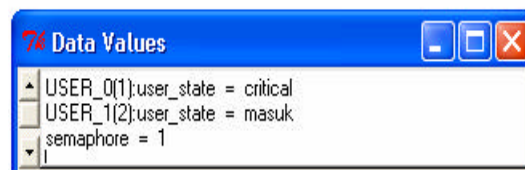
Gambar 4.31 Awal Simulasi



```
74 Simulation Output
777: proc 1 (USER_0) line 35 "pan_in" (state 18) [.(goto)]
778: proc 2 (USER_1) line 78 "pan_in" (state 25) [.(goto)]
779: proc 2 (USER_1) line 46 "pan_in" (state 24) [[[user_state==idle]]]
780: proc 2 (USER_1) line 49 "pan_in" (state 4) [user_state = idle]
781: proc 2 (USER_1) line 54 "pan_in" (state 5) [.(goto)]
782: proc 2 (USER_1) line 78 "pan_in" (state 25) [.(goto)]
783: proc 2 (USER_1) line 46 "pan_in" (state 24) [[[user_state==idle]]]
784: proc 2 (USER_1) line 49 "pan_in" (state 4) [user_state = masuk]
785: proc 2 (USER_1) line 54 "pan_in" (state 5) [.(goto)]
Single Step Run Save in: sim.out Clear Cancel
```

Gambar 4.32 Sebagian Simulasi

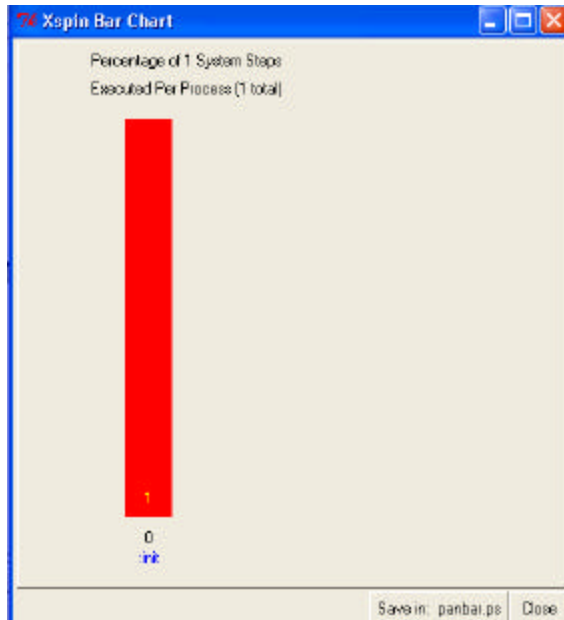
c. Snapshot panel Data value :



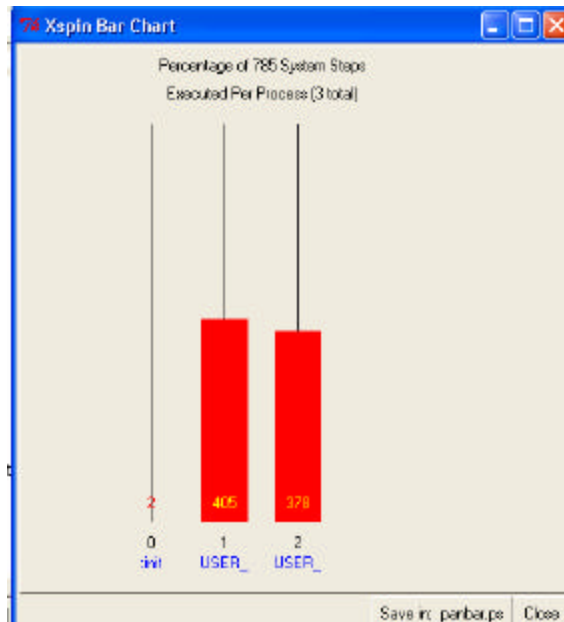
```
74 Data Values
USER_0(1):user_state = critical
USER_1(2):user_state = masuk
semaphore = 1
```

Gambar 4.33 Sebagian Data Value

d. Snapshot panel execution bar chart



Gambar 4.34 Awal execution bar chart



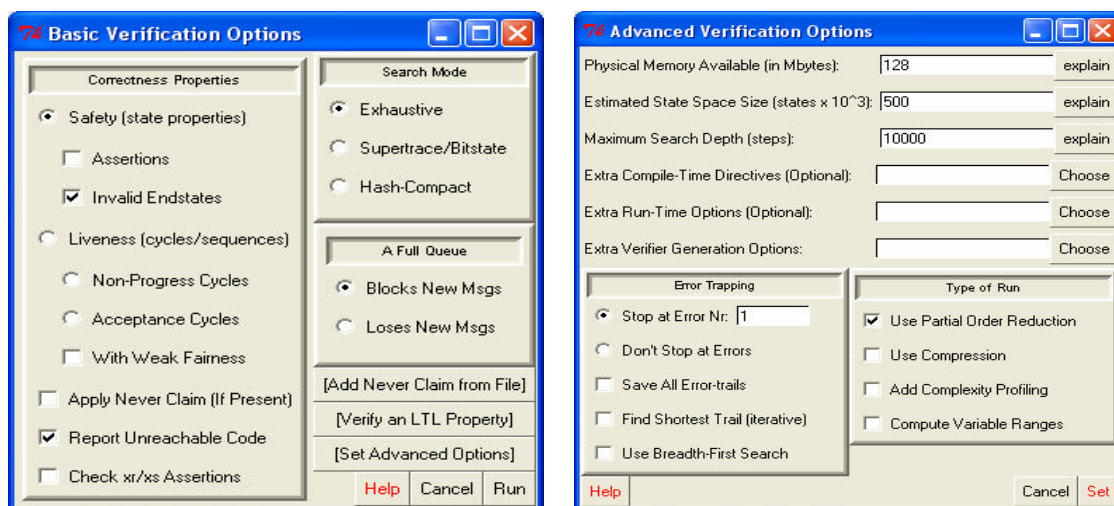
Gambar 4.35 Sebagian execution bar chart

Dari simulasi dapat dilihat bahwa proses-proses saling bergantian memasuki critical section dengan cara melihat status proses lain yakni idle, masuk, critical, dan keluar.

4.2 Verifikasi

Simulasi yang dilakukan diatas belum dapat menunjukkan *correctness* dari setiap model spesifikasi penyelesaian masalah *mutual exclusion*. Untuk dapat menguji apakah kriteria-kriteria kebenaran (seperti yang telah disebutkan di bagian awal) terpenuhi atau tidak, harus dilakukan verifikasi.

Dengan XSPIN, setting parameter verifikasi dilakukan pada window *Verification Option*, seperti terlihat pada gambar 13. Kalau dirasa masih kurang, dapat dilakukannya setting lanjutan pada window *Advanced Verification Option*.



Gambar 4.36. Window *Basic* dan *Advanced Verification Options*

Verifikasi setiap model penyelesaian masalah mutual exclusion yang dilakukan adalah dari segi :

1. *Safety (state properties)* dengan option *assertion* dan *invalid endstates* dihidupkan.
2. *Liveness (cycles/sequence)* dengan option *non progress cycles* dihidupkan dan *with weak fairness* dihidupkan.

4.2.1 Verifikasi Algoritma 3 Peterson

Verifikasi dari segi *Safety (state properties)* dengan option *assertion* dan *invalid endstates* dihidupkan, mode search *supertrace/bitstate*, didapat verifikasi sebagai berikut :

```

(Spin Version 4.0.7 -- 1 August 2003)
+ Partial Order Reduction

Bit statespace search for:
  never claim      - (none specified)
  assertion violations  +
  cycle checks    - (disabled by -DSAFETY)
  invalid end states +

State-vector 24 byte, depth reached 15, errors: 0
  41 states, stored
   9 states, matched
  50 transitions (= stored+matched)
   1 atomic steps
hash factor: 1.59783e+06 (expected coverage: >= 99.9% on avg.)
(max size 2^26 states)

17.342      memory usage (Mbyte)

unreached in proctype user
  (0 of 6 states)
unreached in proctype :init:
  (0 of 4 states)

```

Dari verifikasi ini dapat dilihat bahwa algoritma peterson tidak bermasalah dalam assertion violation dan tidak ada masalah dalam state yang dibangun (tidak ada proses yang melanggar *mutual exclusion*).

Verifikasi dari segi *Liveness (cycles/sequence)* dengan option *non-progress cycles* dihidupkan, dan *with weak fairness* dihidupkan, mode l search *supertrace/bitstate* didapat verifikasi sebagai berikut :

```

Spin Version 4.0.7 -- 1 August 2003)
+ Partial Order Reduction

Bit statespace search for:
  never claim      +
  assertion violations  + (if within scope of claim)
  non-progress cycles  + (fairness enabled)
  invalid end states  - (disabled by never claim)

State-vector 28 byte, depth reached 30, errors: 0
  173 states, stored
  107 states, matched
  280 transitions (= stored+matched)
   2 atomic steps
hash factor: 385683 (expected coverage: >= 99.9% on avg.)
(max size 2^26 states)

42.468      memory usage (Mbyte)

unreached in proctype user
  (0 of 6 states)
unreached in proctype :init:
  (0 of 4 states)

```

Dari verifikasi ini dapat dilihat bahwa algoritma peterson tidak mengalami *non-progress cycles*, malah terjadi *fairness* dalam memasuki critical section (tidak ada perlombaan untuk memasuki *critical section*).

4.2.2 Verifikasi Algoritma Dekker

Verifikasi dari segi *Safety (state properties)* dengan option *assertion* dan *invalid endstates* dihidupkan, mode search *supertrace/bitstate*, didapat verifikasi sebagai berikut:

```

pan: assertion violated (mutex!=2) (at depth 8)
pan: wrote pan_in.trail
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
       + Partial Order Reduction

Bit statespace search for:
  never claim           - (none specified)
  assertion violations  +
  cycle checks          - (disabled by -
DSAFETY)
  invalid end states   +

State-vector 24 byte, depth reached 16, errors: 1
  102 states, stored
   66 states, matched
  168 transitions (= stored+matched)
   0 atomic steps
hash factor: 651542 (expected coverage: >= 99.9% on
avg.)
(max size 2^26 states)

17.342      memory usage (Mbyte)

```

Dari verifikasi ini dapat dilihat bahwa algoritma *Dekker* mengalami assertion violation (pelanggaran) pada (mutex!=2) (at depth 8) sehingga mutual exclusion tidak dapat ditangani. Untuk lebih jelasnya dapat dilihat dalam hasil simulasi dibawah ini :

Hasil simulasi baru setelah verifikasi :

```

preparing trail, please wait...done
1:   proc 1 (B) line 15 "pan_in" (state 1)   [y = 1]
2:   proc 1 (B) line 16 "pan_in" (state 2)   [turn = A_TURN]
3:   proc      1   (B)   line      17   "pan_in"   (state   3)
   [(((x==0)||((turn==B_TURN))))]
4:   proc 1 (B) line 18 "pan_in" (state 4)   [mutex = (mutex+1)]
5:   proc 0 (A) line 6 "pan_in" (state 1)    [x = 1]
6:   proc 0 (A) line 7 "pan_in" (state 2)    [turn = B_TURN]
7:   proc      0   (A)   line      8   "pan_in"   (state   3)

```

```

      [(((y==0)||turn==A_TURN))]
8:   proc 0 (A) line 9 "pan_in" (state 4)   [mutex = (mutex+1)]
spin: line 24 "pan_in", Error: assertion violated
spin: text of failed assertion: assert((mutex!=2))
#processes: 3
9:   proc 2 (monitor) line 24 "pan_in" (state 1)
9:   proc 1 (B) line 19 "pan_in" (state 5)
9:   proc 0 (A) line 10 "pan_in" (state 5)
3 processes created

```

Verifikasi dari segi *Liveness (cycles/sequence)* dengan option *non-progress cycles* dihidupkan dan *with weak fairness* dihidupkan, mode search *supertrace/bitstate* didapat verifikasi sebagai berikut :

```

pan: assertion violated (mutex!=2) (at depth 17)
pan: wrote pan_in.trail
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
      + Partial Order Reduction

Bit statespace search for:
      never claim          +
      assertion violations + (if within scope of claim)
      non-progress cycles  + (fairness enabled)
      invalid end states   - (disabled by never claim)
State-vector 32 byte, depth reached 33, errors: 1
      280 states, stored
      232 states, matched
      512 transitions (= stored+matched)
      0 atomic steps
hash factor: 238822 (expected coverage: >= 99.9% on avg.)
(max size 2^26 states)
42.468      memory usage (Mbyte)

```

Dari verifikasi ini dapat dilihat bahwa algoritma *Dekker* mengalami assertion violation (pelanggaran) pada (mutex!=2) (at depth 17) sehingga mutual exclusion tidak dapat ditangani.

4.2.3 Verifikasi Algoritma Bakery

Verifikasi dari segi *Safety (state properties)* dengan option *assertion* dan *invalid endstates* dihidupkan, mode search *supertrace/bitstate*, didapat verifikasi sebagai berikut:

```

pan: assertion violated (mutex!=2) (at depth 1541)
pan: wrote pan_in.trail
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
        + Partial Order Reduction

Bit statespace search for:
    never claim           - (not selected)
    assertion violations +
    cycle checks         - (disabled by -
DSAFETY)
    invalid end states   +

State-vector 28 byte, depth reached 1571, errors: 1
    9274 states, stored
    5447 states, matched
    14721 transitions (= stored+matched)
    2 atomic steps
hash factor: 7235.46 (expected coverage: >= 99.9%
on avg.)
(max size 2^26 states)

Stats on memory usage (in Megabytes):
0.297      equivalent memory usage for states
(stored*(State-vector + overhead))
16.777     memory used for hash array (-w26)
0.360     memory used for DFS stack (-m10000)
17.342     total actual memory usage

```

Dari verifikasi ini dapat dilihat bahwa algoritma *Bakery* mengalami assertion violation (pelanggaran) pada (mutex!=2) (at depth 1541) sehingga mutual exclusion tidak dapat ditangani. Untuk lebih jelasnya dapat dilihat dalam hasil simulasi dibawah ini :

Hasil simulasi baru setelah verifikasi :

```

preparing trail, please wait...done
1: no process 1 (state 7)
#processes: 1
1:   proc 0 (:init:) line 17 "pan_in" (state 4)
1 processes created

```

Verifikasi dari segi *Liveness (cycles/sequence)* dengan option *non-progress cycles* dihidupkan dan *with weak fairness* dihidupkan, mode l search *supertrace/bitstate* didapat verifikasi sebagai berikut :

```

error: max search depth too small
pan: non-progress cycle (at depth 6318)
pan: wrote pan_in.trail
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
        + Partial Order Reduction

Bit statespace search for:

```

```

        never claim          +
        assertion violations  + (if within scope of claim)
        non-progress cycles  + (fairness enabled)
        invalid end states   - (disabled by never claim)

State-vector 32 byte, depth reached 9999, errors: 1
  7544 states, stored
  3898 states, matched
 11442 transitions (= stored+matched)
    2 atomic steps
hash factor: 8894.48 (expected coverage: >= 99.9% on avg.)
(max size 2^26 states)

Stats on memory usage (in Megabytes):
0.332      equivalent memory usage for states
(stored*(State-vector + overhead))
8.389      memory used for hash array (-w26)
33.554     memory used for bit stack
0.320     memory used for DFS stack (-m10000)
42.570     total actual memory usage

```

Dari verifikasi ini dapat dilihat bahwa algoritma *Bakery* mengalami error karena proses pencarian maksimal terlalu sedikit (error: max search depth too small) dan terjadi non-progress cycle (at depth 6318), sehingga terjadi pelanggaran *mutual exclusion* (tidak ada proses 1 pada state 7) . Untuk lebih jelasnya dapat dilihat dalam hasil simulasi dibawah ini :

Hasil simulasi baru setelah verifikasi :

```

preparing trail, please wait...done
1: no process 1 (state 7)
#processes: 1
1:   proc 0 (:init:) line 17 "pan_in" (state 4)
1 processes created

```

4.2.4 Verifikasi Mutual Exclusion Umum

Verifikasi dari segi *Safety (state properties)* dengan option *assertion* dan *invalid endstates* dihidupkan, mode search supertrace/bitstate, didapat verifikasi sebagai berikut:

```

(Spin Version 4.0.7 -- 1 August 2003)
+ Partial Order Reduction

Bit statespace search for:
  never claim          - (not selected)
  assertion violations  +
  cycle checks         - (disabled by -DSAFETY)
  invalid end states   +

State-vector 36 byte, depth reached 13, errors: 0
  11 states, stored
  11 states, matched

```

```

        22 transitions (= stored+matched)
        13 atomic steps
hash factor: 5.59241e+06 (expected coverage: >= 99.9%
on avg.)
(max size 2^26 states)

17.342      memory usage (Mbyte)

unreached in proctype A
  line 34, state 18, "-end-"
  (1 of 18 states)
unreached in proctype B
  line 57, state 18, "-end-"
  (1 of 18 states)
unreached in proctype :init:
  (0 of 4 states)

```

Dari verifikasi ini dapat dilihat bahwa model ini tidak bermasalah dalam *assertion violation* dan tidak ada masalah dalam state yang dibangun (tidak ada proses yang melanggar *mutual exclusion*).

Verifikasi dari segi *Liveness (cycles/sequence)* dengan option *non-progress cycles* dihidupkan, mode search *supertrace/bitstate* didapat verifikasi sebagai berikut :

```

pan: non-progress cycle (at depth 3)
pan: wrote pan_in.trail
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
  + Partial Order Reduction

Bit statespace search for:
  never claim          +
  assertion violations + (if within scope of claim)
  non-progress cycles  + (fairness disabled)
  invalid end states   - (disabled by never claim)

State-vector 40 byte, depth reached 22, errors: 1
  10 states, stored
   5 states, matched
  15 transitions (= stored+matched)
   7 atomic steps
hash factor: 6.10081e+06 (expected coverage: >= 99.9% on
avg.)
(max size 2^26 states)

42.468      memory usage (Mbyte)

```

Dari verifikasi ini dapat dilihat bahwa model ini mengalami *non-progress cycle (at depth 3)*, sehingga terjadi pelanggaran *mutual exclusion*, dimana dengan adanya *non-progress cycles*, prinsip *fairness* dalam memasuki critical section tidak ada . Untuk lebih jelasnya dapat dilihat dalam hasil simulasi dibawah ini :

Hasil simulasi baru setelah verifikasi :

```
preparing trail, please wait...done
spin: warning, "pan_in", global, 'bit Fail' variable is never used
spin: warning, "pan_in", global, 'int x' variable is never used
spin: warning, "pan_in", global, 'int y' variable is never used
1: no process 1 (state 7)
#processes: 1
1: proc 0 (:init:) line 60 "pan_in" (state 3)
1 processes created
```

Verifikasi dengan penambahan *with weak fairness* dihidupkan, maka verifikasi di dapat sebagai berikut :

```
error: p.o. reduction not compatible with fairness (-f) in models
with rendezvous operations: recompile with -DNOREDUCE
```

Dari verifikasi ini dapat dilihat bahwa operasi *rendevous* model ini tidak sesuai dengan prinsip *fairness* dalam model.

4.2.5 Verifikasi Dijkstra Semaphore

Verifikasi dari segi *Safety (state properties)* dengan option *assertion* dan *invalid endstates* dihidupkan, mode search *supertrace/bitstate*, didapat verifikasi sebagai berikut:

```
(Spin Version 4.0.7 -- 1 August 2003)
+ Partial Order Reduction

Bit statespace search for:
  never claim                - (not selected)
  assertion violations        +
  cycle checks                - (disabled by -DSAFETY)
  invalid end states          +

State-vector 32 byte, depth reached 12, errors: 0
  17 states, stored
  4 states, matched
  21 transitions (= stored+matched)
  0 atomic steps
hash factor: 3.72827e+06 (expected coverage: >= 99.9%
on avg.)
(max size 2^26 states)

17.342      memory usage (Mbyte)

unreached in proctype dijkstra
  line 11, state 10, "-end-"
  (1 of 10 states)
unreached in proctype user
```

```
line 20, state 6, "-end-"
(1 of 6 states)
unreached in proctype :init:
(0 of 5 states)
```

Dari verifikasi ini dapat dilihat bahwa model ini tidak bermasalah dalam *assertion violation* dan tidak ada masalah dalam state yang dibangun (tidak ada proses yang melanggar *mutual exclusion*).

Verifikasi dari segi *Liveness (cycles/sequence)* dengan option *non-progress cycles* dihidupkan, mode search *supertrace/bitstate* didapat verifikasi sebagai berikut :

```
Pan: non-progress cycle (at depth 10)
pan: wrote pan_in.trail
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
       + Partial Order Reduction

Bit statespace search for:
  never claim           +
  assertion violations  + (if within scope of claim)
  non-progress cycles   + (fairness disabled)
  invalid end states    - (disabled by never claim)

State-vector 36 byte, depth reached 23, errors: 1
  12 states, stored
   2 states, matched
  14 transitions (= stored+matched)
   0 atomic steps
hash factor: 5.16222e+06 (expected coverage: >= 99.9% on
avg.)
(max size 2^26 states)

42.468      memory usage (Mbyte)
```

Dari verifikasi ini dapat dilihat bahwa model ini mengalami *non-progress cycle (at depth 10)*, sehingga terjadi pelanggaran *mutual exclusion*, dimana dengan adanya *non-progress cycles*, prinsip *fairness* dalam memasuki critical section tidak ada. Untuk lebih jelasnya dapat dilihat dalam hasil simulasi dibawah ini :

Hasil simulasi baru setelah verifikasi :

```
preparing trail, please wait...done
spin: couldn't find claim (ignored)
 2:   proc 0 (:init:) line 22 "pan_in" (state 1)   [(run dijkstra())]
 4:   proc 1 (dijkstra) line 8 "pan_in" (state 1) [(count==1)]
 6:   proc 0 (:init:) line 22 "pan_in" (state 2)   [(run user())]
 8:   proc 0 (:init:) line 23 "pan_in" (state 3)   [(run user())]
10:   proc 0 (:init:) line 23 "pan_in" (state 4)   [(run user())]
<<<<<START OF CYCLE>>>>>
```

```

12:   proc 1 (dijkstra) line 8 "pan_in" (state -) [values: 1!0]
12:   proc 1 (dijkstra) line 8 "pan_in" (state 2) [sema!0]
13:   proc 4 (user) line 15 "pan_in" (state -) [values: 1?0]
13:   proc 4 (user) line 15 "pan_in" (state 1) [sema?0]
15:   proc 1 (dijkstra) line 8 "pan_in" (state 3) [count = 0]
17:   proc 1 (dijkstra) line 9 "pan_in" (state 4) [((count==0))]
19:   proc 4 (user) line 17 "pan_in" (state -) [values: 1!1]
19:   proc 4 (user) line 17 "pan_in" (state 2) [sema!1]
20:   proc 1 (dijkstra) line 9 "pan_in" (state -) [values: 1?1]
20:   proc 1 (dijkstra) line 9 "pan_in" (state 5) [sema?1]
22:   proc 1 (dijkstra) line 9 "pan_in" (state 6) [count = 1]
24:   proc 1 (dijkstra) line 8 "pan_in" (state 1) [((count==1))]
spin: trail ends after 24 steps
#processes: 5
24:   proc 4 (user) line 14 "pan_in" (state 3)
24:   proc 3 (user) line 14 "pan_in" (state 3)
24:   proc 2 (user) line 14 "pan_in" (state 3)
24:   proc 1 (dijkstra) line 8 "pan_in" (state 2)
24:   proc 0 (:init:) line 24 "pan_in" (state 5)
5 processes created

```

Verifikasi dengan penambahan *with weak fairness* dihidupkan, maka verifikasi di dapat sebagai berikut :

```

error: p.o. reduction not compatible with fairness (-f) in models
      with rendezvous operations: recompile with -DNOREDUCE

```

Dari verifikasi ini dapat dilihat bahwa operasi *rendevous* model ini tidak sesuai dengan prinsip *fairness* dalam model.

4.2.6 Verifikasi Semaphore

Verifikasi dari segi *Safety (state properties)* dengan option *assertion* dan *invalid endstates* dihidupkan, mode search *supertrace/bitstate*, didapat verifikasi sebagai berikut:

```

(Spin Version 4.0.7 -- 1 August 2003)
  + Partial Order Reduction

Bit statespace search for:
  never claim           - (not selected)
  assertion violations  +
  cycle checks          - (disabled by -DSAFETY)
  invalid end states    +

State-vector 20 byte, depth reached 15, errors: 0
  13 states, stored
  25 states, matched
  38 transitions (= stored+matched)
  13 atomic steps
hash factor: 4.79349e+06 (expected coverage: >= 99.9%
on avg.)

```

```

(max size 2^26 states)

17.342      memory usage (Mbyte)

unreached in proctype USER_0
    line 42, state 27, "--end-"
    (1 of 27 states)
unreached in proctype USER_1
    line 78, state 27, "--end-"
    (1 of 27 states)
unreached in proctype :init:
    (0 of 4 states)

```

Dari verifikasi ini dapat dilihat bahwa model ini tidak bermasalah dalam *assertion violation* dan tidak ada masalah dalam state yang dibangun (tidak ada proses yang melanggar *mutual exclusion*).

Verifikasi dari segi *Liveness (cycles/sequence)* dengan option *non-progress cycles* dihidupkan, mode search *supertrace/bitstate* didapat verifikasi sebagai berikut :

```

pan: non-progress cycle (at depth 21)
pan: wrote pan_in.trail
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
    + Partial Order Reduction

Bit statespace search for:
    never claim          +
    assertion violations + (if within scope of claim)
    non-progress cycles  + (fairness disabled)
    invalid end states   - (disabled by never claim)

State-vector 24 byte, depth reached 23, errors: 1
    10 states, stored
     8 states, matched
    18 transitions (= stored+matched)
     7 atomic steps
hash factor: 6.10081e+06 (expected coverage: >= 99.9% on
avg.)
(max size 2^26 states)

42.468      memory usage (Mbyte)

```

Dari verifikasi ini dapat dilihat bahwa model ini mengalami *non-progress cycle* (at depth 10), sehingga terjadi pelanggaran *mutual exclusion*, dimana dengan adanya *non-progress cycles*, prinsip *fairness* dalam memasuki critical section tidak ada. Untuk lebih jelasnya dapat dilihat dalam hasil simulasi dibawah ini

Hasil simulasi baru setelah verifikasi :

```

preparing trail, please wait...done
spin: warning, "pan_in", global, 'bit Fail' variable is never used

```

```

spin: couldn't find claim (ignored)
2:   proc 0 (:init:) line 82 "pan_in" (state 1) [(run USER_0())]
3:   proc 0 (:init:) line 83 "pan_in" (state 2) [(run USER_1())]
5:   proc 2 (USER_1) line 48 "pan_in" (state 1) [((user_state==idle))]
6:   proc 2 (USER_1) line 51 "pan_in" (state 3) [user_state = masuk] <merge 24
now @24>
8:   proc      2   (USER_1)   line      56   "pan_in"   (state   7)
      [(((user_state==masuk)&&!semaphore))] <merge 0 now @8>
8:   proc 2 (USER_1) line 57 "pan_in" (state 8) [user_state = critical] <merge 24
now @9>
8:   proc 2 (USER_1) line 58 "pan_in" (state 9) [semaphore = 1] <merge 24 now @24>
10:  proc 2 (USER_1) line 65 "pan_in" (state 14) [((user_state==critical))]
11:  proc 2 (USER_1) line 68 "pan_in" (state 16) [user_state = keluar]
      <merge 24 now @24>
13:  proc 1 (USER_0) line 12 "pan_in" (state 1) [((user_state==idle))]
14:  proc 1 (USER_0) line 15 "pan_in" (state 3) [user_state = masuk] <merge 24
now @24>
16:  proc 2 (USER_1) line 73 "pan_in" (state 20) [((user_state==keluar))]
      <merge 0 now @21>
16:  proc 2 (USER_1) line 74 "pan_in" (state 21) [user_state = idle]
      <merge 24 now @22>
16:  proc 2 (USER_1) line 75 "pan_in" (state 22) [semaphore = 0] <merge 24
now @24>
18:  proc 2 (USER_1) line 48 "pan_in" (state 1) [((user_state==idle))]
19:  proc 2 (USER_1) line 51 "pan_in" (state 3) [user_state = masuk] <merge 24
now @24>
21:  proc      2   (USER_1)   line      56   "pan_in"   (state   7)
      [(((user_state==masuk)&&!semaphore))] <merge 0 now @8>
21:  proc 2 (USER_1) line 57 "pan_in" (state 8) [user_state = critical] <merge 24
now @9>
21:  proc 2 (USER_1) line 58 "pan_in" (state 9) [semaphore = 1] <merge 24 now @24>
<<<<<START OF CYCLE>>>>>
23:  proc 2 (USER_1) line 65 "pan_in" (state 14) [((user_state==critical))]
24:  proc 2 (USER_1) line 67 "pan_in" (state 15) [user_state = critical]
      <merge 24 now @24>
spin: trail ends after 24 steps
#processes: 3
24:  proc 2 (USER_1) line 46 "pan_in" (state 24)
24:  proc 1 (USER_0) line 10 "pan_in" (state 24)
24:  proc 0 (:init:) line 85 "pan_in" (state 4)
3 processes created

```

Verifikasi dengan penambahan *with weak fairness* dihidupkan, maka verifikasi di dapat sebagai berikut :

```

pan: non-progress cycle (at depth 62)
pan: wrote pan_in.trail
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
      + Partial Order Reduction

Bit statespace search for:
      never claim          +
      assertion violations + (if within scope of claim)

```

```

    non-progress cycles      + (fairness enabled)
    invalid end states      - (disabled by never claim)

State-vector 28 byte, depth reached 117, errors: 1
    66 states, stored
    101 states, matched
    167 transitions (= stored+matched)
    56 atomic steps
hash factor: 1.00162e+06 (expected coverage: >= 99.9% on
avg.)
(max size 2^26 states)
42.468      memory usage (Mbyte)

```

Dari verifikasi ini dapat dilihat bahwa model ini mengalami non-progress cycle (at depth 62), sehingga terjadi pelanggaran *mutual exclusion*, tetapi prinsip *fairness* dalam memasuki critical section masih ada. Untuk lebih jelasnya dapat dilihat dalam hasil simulasi dibawah ini

Hasil simulasi baru setelah verifikasi :

```

preparing trail, please wait...done
spin: warning, "pan_in", global, 'bit Fail' variable is never used
spin: couldn't find claim (ignored)
 2:  proc 0 (:init:) line 82 "pan_in" (state 1)  [(run USER_0())]
 3:  proc 0 (:init:) line 83 "pan_in" (state 2)  [(run USER_1())]
 5:  proc 2 (USER_1) line 48 "pan_in" (state 1) [((user_state==idle))]
 6:  proc 2 (USER_1) line 50 "pan_in" (state 2) [user_state = idle]      <merge 24
now @24>
 8:  proc 2 (USER_1) line 48 "pan_in" (state 1) [((user_state==idle))]
 9:  proc 2 (USER_1) line 51 "pan_in" (state 3) [user_state = masuk]      <merge 24
now @24>
11:  proc      2      (USER_1)      line      56      "pan_in"      (state      7)
    [(((user_state==masuk)&&! (semaphore)))] <merge 0 now @8>
11:  proc 2 (USER_1) line 57 "pan_in" (state 8) [user_state = critical]      <merge 24
now @9>
11:  proc 2 (USER_1) line 58 "pan_in" (state 9) [semaphore = 1] <merge 24 now @24>
13:  proc 2 (USER_1) line 65 "pan_in" (state 14) [((user_state==critical))]
14:  proc 2 (USER_1) line 68 "pan_in" (state 16) [user_state = keluar]
    <merge 24 now @24>
16:  proc 1 (USER_0) line 12 "pan_in" (state 1) [((user_state==idle))]
17:  proc 1 (USER_0) line 14 "pan_in" (state 2) [user_state = idle]      <merge 24
now @24>
19:  proc 2 (USER_1) line 73 "pan_in" (state 20) [((user_state==keluar))]
    <merge 0 now @21>
19:  proc 2 (USER_1) line 74 "pan_in" (state 21) [user_state = idle]
    <merge 24 now @22>
19:  proc 2 (USER_1) line 75 "pan_in" (state 22) [semaphore = 0] <merge 24
now @24>

```

```

21:   proc 2 (USER_1) line 48 "pan_in" (state 1) [((user_state==idle))]
22:   proc 2 (USER_1) line 51 "pan_in" (state 3) [user_state = masuk]      <merge 24
now @24>
24:   proc          2   (USER_1)   line          56   "pan_in"   (state   7)
[(((user_state==masuk)&&! (semaphore)))] <merge 0 now @8>
24:   proc 2 (USER_1) line 57 "pan_in" (state 8) [user_state = critical]    <merge 24
now @9>
24:   proc 2 (USER_1) line 58 "pan_in" (state 9) [semaphore = 1] <merge 24 now @24>
26:   proc 1 (USER_0) line 12 "pan_in" (state 1) [((user_state==idle))]
27:   proc 1 (USER_0) line 15 "pan_in" (state 3) [user_state = masuk]      <merge 24
now @24>
29:   proc 2 (USER_1) line 65 "pan_in" (state 14)      [((user_state==critical))]
30:   proc 2 (USER_1) line 68 "pan_in" (state 16)      [user_state = keluar]
<merge 24 now @24>
32:   proc 2 (USER_1) line 73 "pan_in" (state 20)      [((user_state==keluar))]
<merge 0 now @21>
32:   proc 2 (USER_1) line 74 "pan_in" (state 21)      [user_state = idle]
<merge 24 now @22>
32:   proc 2 (USER_1) line 75 "pan_in" (state 22)      [semaphore = 0] <merge 24
now @24>
34:   proc 2 (USER_1) line 48 "pan_in" (state 1) [((user_state==idle))]
35:   proc 2 (USER_1) line 51 "pan_in" (state 3) [user_state = masuk]      <merge 24
now @24>
37:   proc          1   (USER_0)   line          20   "pan_in"   (state   7)
[(((user_state==masuk)&&! (semaphore)))] <merge 0 now @8>
37:   proc 1 (USER_0) line 21 "pan_in" (state 8) [user_state = critical]    <merge 24
now @9>
37:   proc 1 (USER_0) line 22 "pan_in" (state 9) [semaphore = 1] <merge 24 now @24>
39:   proc 1 (USER_0) line 29 "pan_in" (state 14)      [((user_state==critical))]
40:   proc 1 (USER_0) line 32 "pan_in" (state 16)      [user_state = keluar]
<merge 24 now @24>
42:   proc          2   (USER_1)   line          61   "pan_in"   (state  11)
[(((user_state==masuk)&&semaphore))] <merge 0 now @12>
42:   proc 2 (USER_1) line 62 "pan_in" (state 12)      [user_state = masuk]
44:   proc 1 (USER_0) line 37 "pan_in" (state 20)      [((user_state==keluar))]
<merge 0 now @21>
44:   proc 1 (USER_0) line 38 "pan_in" (state 21)      [user_state = idle]
<merge 24 now @22>
44:   proc 1 (USER_0) line 39 "pan_in" (state 22)      [semaphore = 0] <merge 24
now @24>
46:   proc 1 (USER_0) line 12 "pan_in" (state 1) [((user_state==idle))]
47:   proc 1 (USER_0) line 15 "pan_in" (state 3) [user_state = masuk]      <merge 24
now @24>
49:   proc          2   (USER_1)   line          56   "pan_in"   (state   7)
[(((user_state==masuk)&&! (semaphore)))] <merge 0 now @8>
49:   proc 2 (USER_1) line 57 "pan_in" (state 8) [user_state = critical]    <merge 24
now @9>
49:   proc 2 (USER_1) line 58 "pan_in" (state 9) [semaphore = 1] <merge 24 now @24>
51:   proc 2 (USER_1) line 65 "pan_in" (state 14)      [((user_state==critical))]
52:   proc 2 (USER_1) line 68 "pan_in" (state 16)      [user_state = keluar]
<merge 24 now @24>
54:   proc 2 (USER_1) line 73 "pan_in" (state 20)      [((user_state==keluar))]
<merge 0 now @21>
54:   proc 2 (USER_1) line 74 "pan_in" (state 21)      [user_state = idle]
<merge 24 now @22>
54:   proc 2 (USER_1) line 75 "pan_in" (state 22)      [semaphore = 0] <merge 24
now @24>
56:   proc          1   (USER_0)   line          20   "pan_in"   (state   7)
[(((user_state==masuk)&&! (semaphore)))] <merge 0 now @8>

```

```

56:   proc 1 (USER_0) line 21 "pan_in" (state 8) [user_state = critical]   <merge 24
now @9>
56:   proc 1 (USER_0) line 22 "pan_in" (state 9) [semaphore = 1] <merge 24 now @24>
58:   proc 1 (USER_0) line 29 "pan_in" (state 14)      [((user_state==critical))]
59:   proc 1 (USER_0) line 32 "pan_in" (state 16)      [user_state = keluar]
<merge 24 now @24>
61:   proc 2 (USER_1) line 48 "pan_in" (state 1) [((user_state==idle))]
62:   proc 2 (USER_1) line 50 "pan_in" (state 2) [user_state = idle]      <merge 24
now @24>
<<<<<START OF CYCLE>>>>
64:   proc 2 (USER_1) line 48 "pan_in" (state 1) [((user_state==idle))]
65:   proc 2 (USER_1) line 50 "pan_in" (state 2) [user_state = idle]      <merge 24
now @24>
67:   proc 2 (USER_1) line 48 "pan_in" (state 1) [((user_state==idle))]
68:   proc 2 (USER_1) line 51 "pan_in" (state 3) [user_state = masuk]    <merge 24
now @24>
70:   proc 1 (USER_0) line 37 "pan_in" (state 20)      [((user_state==keluar))]
<merge 0 now @21>
70:   proc 1 (USER_0) line 38 "pan_in" (state 21)      [user_state = idle]
<merge 24 now @22>
70:   proc 1 (USER_0) line 39 "pan_in" (state 22)      [semaphore = 0] <merge 24
now @24>
72:   proc      2      (USER_1)      line      56      "pan_in"      (state      7)
[(((user_state==masuk)&&!semaphore))] <merge 0 now @8>
72:   proc 2 (USER_1) line 57 "pan_in" (state 8) [user_state = critical] <merge 24
now @9>
72:   proc 2 (USER_1) line 58 "pan_in" (state 9) [semaphore = 1] <merge 24 now @24>
74:   proc 2 (USER_1) line 65 "pan_in" (state 14)      [((user_state==critical))]
75:   proc 2 (USER_1) line 68 "pan_in" (state 16)      [user_state = keluar]
<merge 24 now @24>
77:   proc 2 (USER_1) line 73 "pan_in" (state 20)      [((user_state==keluar))]
<merge 0 now @21>
77:   proc 2 (USER_1) line 74 "pan_in" (state 21)      [user_state = idle]
<merge 24 now @22>
77:   proc 2 (USER_1) line 75 "pan_in" (state 22)      [semaphore = 0] <merge 24
now @24>
79:   proc 1 (USER_0) line 12 "pan_in" (state 1) [((user_state==idle))]
80:   proc 1 (USER_0) line 15 "pan_in" (state 3) [user_state = masuk]    <merge 24
now @24>
82:   proc 2 (USER_1) line 48 "pan_in" (state 1) [((user_state==idle))]
83:   proc 2 (USER_1) line 51 "pan_in" (state 3) [user_state = masuk]    <merge 24
now @24>
85:   proc      2      (USER_1)      line      56      "pan_in"      (state      7)
[(((user_state==masuk)&&!semaphore))] <merge 0 now @8>
85:   proc 2 (USER_1) line 57 "pan_in" (state 8) [user_state = critical] <merge 24
now @9>
85:   proc 2 (USER_1) line 58 "pan_in" (state 9) [semaphore = 1] <merge 24 now @24>
87:   proc 2 (USER_1) line 65 "pan_in" (state 14)      [((user_state==critical))]
88:   proc 2 (USER_1) line 68 "pan_in" (state 16)      [user_state = keluar]
<merge 24 now @24>
90:   proc 2 (USER_1) line 73 "pan_in" (state 20)      [((user_state==keluar))]
<merge 0 now @21>
90:   proc 2 (USER_1) line 74 "pan_in" (state 21)      [user_state = idle]
<merge 24 now @22>
90:   proc 2 (USER_1) line 75 "pan_in" (state 22)      [semaphore = 0] <merge 24
now @24>
92:   proc 2 (USER_1) line 48 "pan_in" (state 1) [((user_state==idle))]
93:   proc 2 (USER_1) line 50 "pan_in" (state 2) [user_state = idle]    <merge 24
now @24>

```

```

95:   proc      1  (USER_0)  line      20  "pan_in"  (state  7)
      [(((user_state==masuk)&&!(semaphore)))] <merge 0 now @8>
95:   proc  1 (USER_0) line  21 "pan_in" (state 8) [user_state = critical]    <merge  24
now @9>
95:   proc  1 (USER_0) line  22 "pan_in" (state 9) [semaphore = 1] <merge 24 now @24>
97:   proc  1 (USER_0) line  29 "pan_in" (state 14)      [((user_state==critical))]
98:   proc  1 (USER_0) line  32 "pan_in" (state 16)      [user_state      =      keluar]
      <merge 24 now @24>
100:  proc  2 (USER_1) line  48 "pan_in" (state 1) [((user_state==idle))]
101:  proc  2 (USER_1) line  50 "pan_in" (state 2) [user_state = idle]      <merge  24
now @24>
spin: trail ends after 102 steps
#processes: 3
102:  proc  2 (USER_1) line  46 "pan_in" (state 24)
102:  proc  1 (USER_0) line  10 "pan_in" (state 24)
102:  proc  0 (:init:) line  85 "pan_in" (state 4)
3 processes created

```

BAB V KESIMPULAN

Setelah melakukan simulasi dan verifikasi terhadap model penyelesaian masalah mutual exclusion, dapat diambil beberapa kesimpulan sebagai berikut :

1. Bila simulasi yang dilakukan berhasil, belum tentu bahwa model tersebut diverifikasi kerjanya sudah benar.
2. Beberapa model mengalami pelanggaran *assertion violations* seperti model algoritma Dekker dan Bakery.
3. Beberapa model mengalami pelanggaran *non-progress cycle* seperti Model algoritma Bakery, Mutual exclusion umum, Dijkstra semaphore, semaphore.
4. Model Peterson tidak didapati pelanggaran *mutual exclusion*.
5. Dari verifikasi yang dilakukan bahwa penanganan mutual exclusion pada proses kongkurensi sangat penting, karena jika tidak dapat mengakibatkan deadlock dan starvation.

DAFTAR PUSTAKA

1. Hariyanto, Bambang. Sistem Operasi. Edisi 2. Bandung. Informatika. 1999.
2. Kusumadewi, Sri. Sistem Operasi. Yogyakarta, J & J Learning, 2000
3. Tenenbaum, Andrew S., "Modern Operating System", Englewood Cliffs, New Jersey : Prentice-Hall Inc., 1992.
4. stalling, William, "Operating System", 2nd Englewood Cliffs, New Jersey : Prentice-Hall Inc., 1995.