VERIFIKASI ALGORITMA *DISTRIBUTED MUTUAL EXCLUSION* DENGAN PROMELA/SPIN

Skripsi

Oleh :

RUSDIYANTO

H 111 06 026

JURUSAN MATEMATIKA

FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM

UNIVERSITAS HASANUDDIN

MAKASSAR

2011

"VERIFIKASI ALGORITMA *DISTRIBUTED MUTUAL EXCLUSION* DENGAN PROMELA/SPIN"

S K R I P S I

Diajukan sebagai salah satu syarat untuk memperoleh gelar Sarjana Sains pada Jurusan Matematika Fakultas Matematika dan Ilmu Pengetahuan Alam Universitas Hasanuddin Makassar

Oleh:

R U S D I Y A N T O

H 111 06 026

JURUSAN MATEMATIKA

FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM

UNIVERSITAS HASANUDDIN

MAKASSAR

2011

P E R N Y A T A A N

Saya yang bertanda tangan di bawah ini menyatakan dengan sesungguh-sungguhnya bahwa skripsi yang saya buat dengan judul :

"VERIFIKASI ALGORITMA *DISTRIBUTED MUTUAL EXCLUSION* DENGAN PROMELA/SPIN"

adalah benar hasil kerja saya sendiri, bukan hasil plagiat dan belum pernah dipublikasikan dalam bentuk apapun.

Makassar, 16 Agustus 2011

RUSDIYANTO
NIM : H 111 06 026

"VERIFIKASI ALGORITMA *DISTRIBUTED MUTUAL EXCLUSION* DENGAN PROMELA/SPIN"

Disetujui Oleh :

Pembimbing Utama	**Pembimbing Pertama**
Dr. Armin Lawi, M.Eng.	**Andi Galsan Mahie, S.Si, M.Si.**
NIP. 19720423 199512 1 001	**NIP. 19720628 200501 1 002**

Pembimbing Kedua

Hendra, S.Si, M.Kom.
NIP. 19760102 200312 1 001

Pada tanggal : 16 Agustus 2011

Pada hari ini, Selasa tanggal 16 Agustus 2011, Panitia Ujian Skripsi menerima dengan baik skripsi yang berjudul :

"VERIFIKASI ALGORITMA *DISTRIBUTED MUTUAL EXCLUSION* DENGAN PROMELA/SPIN"

yang diajukan untuk memenuhi salah satu syarat guna memperoleh gelar Sarjana Sains pada Program Studi Matematika Jurusan Matematika Fakultas Matematika dan Ilmu Pengetahuan Alam Universitas Hasanuddin.

Makassar, 16 Agustus 2011

PANITIA UJIAN SKRIPSI

		Tanda Tangan
1. Ketua	: **Drs. Diaraya, M.Ak.**	(..................................)
2. Sekretaris	: **Drs. H. Muhammad Hasbi, M.Sc.**	(..................................)
3. Anggota	: **Dr. Armin Lawi, M.Eng.**	(..................................)
4. Anggota	: **Andi Galsan Mahie, S.Si, M.Si.**	(..................................)
5. Anggota	: **Hendra, S.Si, M.Kom.**	(..................................)

KATA PENGANTAR

Puji syukur kehadirat Allah SWT yang telah memberikan kasih sayang serta kekuatan kepada penulis, yang membuat segala hal yang tidak mungkin menjadi mungkin dan yang membuat sulit menjadi mudah. Sujud syukurku atas nikmat dan rizkiMu, karena berkat rahmat, hidayah, bimbingan serta kehendak-Nya skripsi ini dapat terselesaikan walaupun dalam bentuk sederhana.

Skripsi ini diajukan sebagai salah satu syarat ketentuan akademik sebagai tugas akhir guna meraih gelar sarjana di Fakultas Matematika dan Ilmu Pengetahuan Alam Universitas Hasanuddin.

Terwujudnya skripsi ini tidaklah mudah, begitu penuh berbagai rintangan, tantangan dan hambatan yang harus penulis lewati dengan penuh kesabaran dalam proses penyusunannya. Oleh karena itu dengan penuh ketulusan, keikhlasan dan rasa hormat penulis menyatakan penghargaan dan ucapan terima kasih yang tak terhingga kepada pihak yang mengulurkan tangan membantu penulis selama mengikuti pendidikan sampai penyelesaian skripsi ini antara lain :

1. Kedua orang tuaku Ayahanda **Alm. Syahlan Sinring** dan Ibunda **Sitti Maryam** yang telah membesarkan dan mendidik penulis dengan penuh kesabaran dan dengan limpahan cinta dan kasih sayang, do’a yang tulus dan nikmat rizki dari setiap tetes keringat yang dikeluarkan, semoga apa yang ananda lakukan dapat menjadi kebanggaan bagi kedua orang tuaku tercinta.

2. Kakakku **Idayanti Syahlan** dan suaminya **Arief Rahim,** adik-adikku **Puspa Amelia, Muhammad Furqan** dan **Qarina Syahrani** atas segala kebersamaan, bantuan, dan motivasi kepada penulis.
3. Bapak **Dr. Armin Lawi, M.Eng** sebagai dosen pembimbing utama, terima kasih atas ilmu, koreksi penulisan, kesabaran dan waktu yang telah diluangkan untuk penulis. Bapak **Andi Galsan Mahie, S.Si. M.Si** sebagai pembimbing pertama, terima kasih atas bimbingan ilmu, saran dan waktunya. Bapak **Hendra, S.Si, M.Kom** sebagai pembimbing kedua, terima kasih atas koreksi, waktu dan bimbingannya.
4. Bapak **Drs. Diaraya, S.Si. M.Ak** sebagai pembimbing akademik sekaligus dosen penguji serta Bapak **Drs. H. Muhammad Hasbi, S.Si, M.Sc** sebagai dosen penguji, terima kasih atas segala masukan, kritik, dan saran sehingga penulis dapat menyelesaikan skripsi ini sesuai dengan yang diharapkan.
5. Bapak **Drs. Khaeruddin, M.Sc** selaku Ketua Jurusan Matematika atas arahan dan nasehat selama penulis menjadi mahasiswa di Jurusan Matematika**, Bapak Andi Kresna Jaya, S.Si, M.Si** selaku Sekretaris Jurusan Matematika, dan para **Dosen Jurusan Matematika** yang telah memberikan bekal ilmu dan pengetahuan yang tiada harga.
6. Para **Staf Jurusan Matematika, Pak Nasir** dan **Pak Sutamin, S.Sos,** yang telah memberikan bantuan dan kerjasamanya selama penulis menjalani perkuliahan sampai ujian akhir
7. Buat teman – teman angkatanku Matematika‘06 atas segala bantuan dan kebersamaannya selama menuntut ilmu.

8. Buat kanda –kanda senior dan adik – adikku angkatan **‘07, ‘08, ‘09, ’10** dan seluruh **warga Himatika** yang tidak dapat penulis sebutkan satu per satu.

Tiada kemampuan penulis untuk membalas semua bantuan dan pertolongan yang telah diberikan, selain seuntai do’a dan harapan, kiranya uluran tangan yang tulus Bapak/Ibu, saudara – saudaraku , sahabat – sahabatku serta pihak – pihak yang telah membantu penulis, semoga mendapat balasan pahala yang berlipat ganda dari Allah SWT.

Dan di akhir kata dengan segala kerendahan hati, penulis persembahkan skripsi ini. Semoga skripsi ini dapat bermanfaat untuk menambah wawasan dan pengetahuan bagi pihak – pihak yang berkepentingan, serta penulis sendiri. Dan penulis menyadari sepenuhnya “Tiada hasil tanpa usaha dan do’a”. Demikian pula skripsi ini, terdapat banyak kekurangan baik dari segi kualitas maupun kuantitasnya. Untuk itu penulis sangat menghargai setiap masukan dan koreksi yang konstruktif demi penyempurnaan skripsi ini.

Makassar, 16 Agustus 2011

Penulis

ABSTRAK

Beberapa sumber daya *(resource)* pada sistem terdistribusi, hanya boleh diakses secara *mutually exclusive* untuk mencapai hasil yang diinginkan bersama. Akses inilah yang menyebabkan terjadinya masalah *mutual exclusion*; akses dimana sumber daya tersebut tidak dapat dieksekusi sekaligus oleh proses yang banyak dalam waktu yang bersamaan. Jadi proses-proses kongkuren *(concurrent processes)* yang kondisinya pada saat bersamaan melakukan *share* terhadap sumber daya membutuhkan algoritma *mutual exclusion* untuk menjamin hanya satu proses yang mengakses sumber daya pada satu waktu tertentu.

Beberapa algoritma terdistribusi *(distributed algorithms)* dapat menyelesaikan masalah ini dengan prinsip kerja yang juga berbeda. Dalam bahasan ini akan membahas bagaimana algoritma tersebut dibuktikan secara matematis, dan juga dengan verifikasi menggunakan tools SPIN *(Simple Promela Interpreter)* yang dimodelkan menggunakan bahasa *PROMELA (Protocol/Process Meta Language*) yang menyerupai bahasa C.

Kata Kunci : *Mutual Exclusion*, *Critical Section,* Algoritma Terdistribusi PROMELA, SPIN, Proses Kongkurensi.

ABSTRACT

Some resources in distributed systems, may only be accessed mutually exclusive to achieve the desired results together. Access is what causes the problem of mutual exclusion, access to which resources can not be executed at once by a process that many at the same time. So concurrent processes whose condition is at the same time to share the resources needed to ensure mutual exclusion algorithm is only one process is accessing the resource at a particular time.

Several distributed algorithms can solve this problem by working principles are also different. In this discussion will discuss how the algorithm is proved mathematically, and also by using the verification tools SPIN (Simple Promela

Interpreter) are modeled using the language Promela (Protocol / Process Meta Language), which resembles the C language.
Keywords : Mutual Exclusion, Critical Section, Distributed Algorithms PROMELA, SPIN, Concurrent Processes.

DAFTAR ISI

HALAMAN JUDUL

LEMBAR KEOTENTIKAN ii

KATA PENGANTAR vi

ABSTRAK ix

ABSTRACT x

DAFTAR ISI xi

DAFTAR GAMBAR xiii

BAB I PENDAHULUAN

I.1 Latar Belakang 1

I.2 Rumusan Masalah 3

I.3 Batasan Masalah 4

I.4 Tujuan Penulisan 4

I.5 Sistematika Penulisan 4

BAB II TINJAUAN PUSTAKA

II.1 Sistem Terdistribusi 5

II.2 Mutual Exclusion 7

II.3 Algoritma *Distributed Mutual Exclusion* 11

II.3.1 Algoritma I 13

II.3.2 Algoritma II 14

II.3.3 Algoritma III (Peterson) 16

II.3.4 Algoritma Dijkstra Semaphore 18

II.4 PROMELA dan SPIN 19

II.4.1 Promela 20

II.4.2 Spin 23

II.5 Alur Kerja 25

BAB III ALGORITMA *DISTRIBUTED MUTUAL EXCLUSION*

III.1 Algoritma Peterson 26

III.2 Algoritma Dijkstra Semaphore 28

BAB IV HASIL DAN PEMBAHASAN

IV.1 Spesifikasi Rancangan Algoritma 32

IV.1.1 Algoritma Peterson 32

IV.1.2 Algoritma Dijkstra Semaphore 33

IV.2 Verifikasi Algoritma 33

IV.2.1 Simulasi 34

IV.2.1.1 Simulasi Algoritma Peterson 35

IV.2.1.2 Simulasi Algoritma Dijkstra Semaphore 38

IV.2.2 Verifikasi 41

IV.2.2.1 Verifikasi Algoritma Peterson 42

IV.2.2.2 Verifikasi Algoritma Dijkstra Semaphore 44

BAB V PENUTUP

V.1 Kesimpulan 48

V.2 Saran 49

DAFTAR PUSTAKA 50

DAFTAR GAMBAR

No.Gambar	Nama Gambar	Hal.
2.1	Sistem Terdistribusi…….................................…………	5
2.2	Model Sistem Terdistribusi................…....………………	6
2.3	Terjadinya Konflik Akses Beberapa Proses..........................	8
2.4	Skenario *Mutual Exlusion*..	9
2.5	Abstraksi Algoritma *Distributed Mutual Exclusion*..............	11
2.6	Alur Kerja...	25
4.1	Window Parameter Simulasi....…...................................……..	34
4.2	Tampilan Akhir MSC...……..	35
4.3	Data Value..	35
4.4	Awal Simulasi…..……...	36
4.5	Akhir Simulasi......................................…..................……	36
4.6	Awal *Execution Bar Chart*.......................................……	37
4.7	Akhir *Execution Bar Chart*...…..	37
4.8	Sebagian Tampilan MSC......................................….….....	38
4.9	Awal Simulasi…...……...	39
4.10	Sebagian Simulasi...…..	39
4.11	Awal *Execution Bar Chart*.......................................……	40
4.12	Sebagian *Execution Bar Chart*...……	40
4.13	Window *Basic Verification Options*…...............................……	41
4.14	Window *Advanced Verification Options*…..........................	42

4.15 Hasil Verifikasi untuk Spesifikasi *Safety Condition*............. 43

4.16 Hasil Verifikasi untuk Spesifikasi *Liveness Condition*.......... 44

4.17 Hasil Verifikasi untuk Spesifikasi *Safety Condition*............. 45

4.18 Hasil Verifikasi untuk Spesifikasi *Liveness Condition*.......... 46

4.19 Hasil Akhir Simulasi baru untuk *Liveness Condition*........... 47

4.20 Hasil Verifikasi untuk Spesifikasi *Liveness Condition*.......... 47

BAB I

PENDAHULUAN

I.1. Latar Belakang

Sistem komputer terdistribusi adalah koleksi komputer *autonomous* yang tidak memiliki memori bersama (atau memori global) dan bekomunikasi hanya dengan mekanisme saling bertukar pesan *(message passing)* menggunakan suatu fasilitas media komunikasi. Sistem terdistribusi berbeda dengan sistem tersentralisasi *(centralized system)*. Dalam sistem tersentralisasi, beberapa komputer terhubung pada suatu host (atau komputer utama) dan metode komunikasi antar komputer terpusat/tersentralisasi berada pada komputer utama tersebut. Namun demikian, baik sistem terdistribusi atau tersentralisasi, masing-masing user pada setiap terminal tidak dapat mengetahui proses yang berlangsung pada sistem operasi.

Perkembangan sistem komputer mendatang adalah menuju ke sistem *multiprocessing*, *multiprogramming*, terdistribusi dan paralel yang mengharuskan adanya proses-proses yang berjalan bersama dalam waktu yang bersamaan. Hal ini merupakan masalah yang perlu perhatian dari perancang sistem operasi. Kongkurensi merupakan landasan umum perancangan sistem operasi. Proses-proses yang kongkuren *(concurrent processes)* adalah proses-proses (lebih dari satu) berada pada saat yang sama. Proses-proses ini dapat sepenuhnya tak bergantung dengan yang lainnya, tapi dapat juga saling berinteraksi. Proses-proses yang berinteraksi memerlukan sinkronisasi agar terkendali dengan baik. Proses-proses kongkuren berkompetisi ketika proses-proses bersaing menggunakan sumber daya *(resource)* yang sama. Dua proses atau lebih perlu mengakses sumber daya yang sama pada suatu saat. Masing-masing proses tidak peduli

keberadaan proses-proses lain dan masing-masing proses tidak dipengaruhi proses-proses lain.

Proses-proses yang melakukan akses secara kongkuren pada data yang digunakan bersama-sama menyebabkan data tidak konsisten *(inconsistence).* Agar data konsisten dibutuhkan mekanisme untuk menjamin eksekusi yang berurutan pada proses-proses yang bekerja sama. Proses-proses kongkuren yang melakukan kerjasama terhadap pemakaian sumber daya *(resource)* atau sekumpulan sumber daya, membutuhkan *mutual exclusion* sebagai syarat untuk memastikan dalam pengaksesan sumber daya. *Mutual exclusion* menjamin hanya satu proses yang mengakses sumber daya pada satu interval waktu tertentu. Bagian program yang sedang mengakses memori atau sumber daya yang dipakai bersama disebut *critical section/region.* Gambaran penting dari sistem adalah, ketika sebuah proses dijalankan di dalam *critical section*, tidak ada proses lain yang diijinkan untuk menjalankan *critical section*-nya. Jadi proses-proses kongkuren yang kondisinya pada saat bersamaan melakukan *share* terhadap sumber daya tidak akan dieksekusi.

Kesuksesan proses-proses kongkuren memerlukan pendefinisian *critical section* dan menjamin *mutual exclusion* diantara proses-proses kongkuren yang sedang berjalan. Karena pentingnya penerapan *mutual exclusion* dalam mengendalikan proses kongkurensi dalam pemrosesannya, sehingga dibutuhkan penanganan proses kongkurensi tersebut agar dapat dikerjakan oleh sumber daya (*resource).*

Terdapat banyak algoritma terdistribusi (*distributed algorithms)* yang dapat menyelesaikan masalah *mutual exclusion* pada sebuah sistem terdistribusi. Prinsip kerja dari algoritma tersebut juga berbeda didalam mengatur penangan masalah ini, sehingga

diperlukan bahasan untuk meninjau kinerjanya. Pembuktian algoritma dapat dilakukan secara matematis, namun dapat pula dilakukan dengan verifikasi menggunakan tools **SPIN** ***(Simple Promela Interpreter)*** yang bersifat sebagai *model checker* pada sistem yang dimodelkan menggunakan bahasa **PROMELA** ***(Protocol/Process Meta Language).*** Bahasa pemodelan ini menyerupai bahasa C (ditambah feature dari CSP).

Berdasarkan uraian di atas, maka penulis ingin mengkajinya dan menuangkannya dalam bentuk skripsi sebagai keperluan tugas akhir dengan judul:

"Verifikasi Algoritma *Distributed Mutual Exclusion* dengan PROMELA/ SPIN".

I.2. Rumusan Masalah

Masalah yang dirumuskan dalam penulisan skripsi ini adalah bagaimana kinerja dari algoritma *distributed mutual exclution* jika terjadi proses kongkurensi pada sistem terdistribusi, serta bagaimana menverifikasinya dengan PROMELA/ SPIN.

I.3. Batasan Masalah

Dalam skripsi ini penulis membatasi masalah hanya memverifikasi algoritma *distributed mutual exclution* dengan PROMELA/SPIN.

I.4. Tujuan Penulisan

Adapun tujuan dari penulisan skripsi ini adalah:

1. Mengkaji masalah *mutual exclution* pada sistem terdistribusi.
2. Meninjau kinerja dari algoritma *distributed mutual exclution* jika terjadi proses kongkurensi.

3. Menverifikasi algoritma *distributed mutual exclution* tersebut dengan PROMELA/ SPIN.

I.5. Sistematika Penulisan

Penulisan tugas akhir ini akan diorganisasikan dalam 5 (empat) bab dengan sistematika penulisan sebagai berikut :

BAB I , akan dibahas mengenai hal – hal yang melatarbelakangi penulisan tugas akhir.

BAB II, akan dibahas beberapa landasan teoritis yang digunakan dalam penulisan.

BAB III, akan dilakukan pembuktian secara matematis pada algoritma yang digunakan.

BAB IV akan dibahas spesifikasi rancangan algoritma dan menverifikasinya.

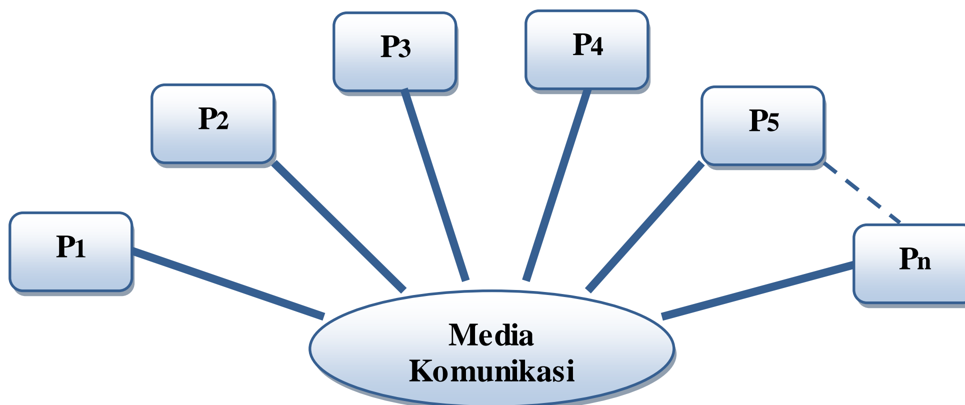
BAB V penutup.

BAB II

TINJAUAN PUSTAKA

II.1. Sistem Terdistribusi

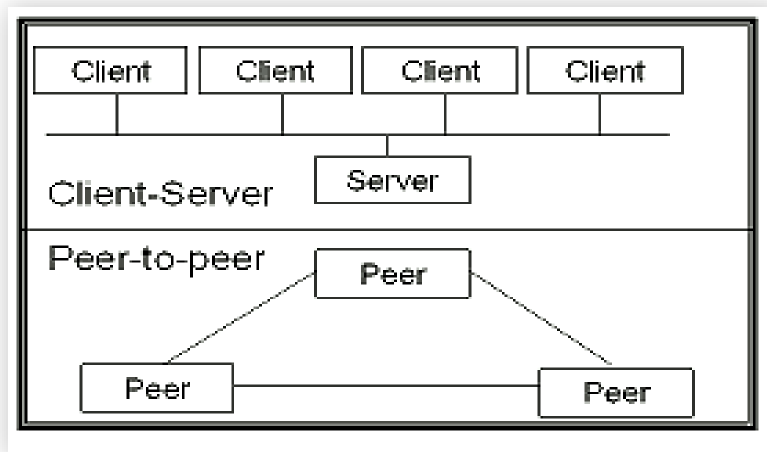
Sistem terdistribusi adalah sekumpulan prosesor yang tidak berbagi memori, atau *clock* dan bekomunikasi hanya dengan mekanisme saling bertukar pesan *(message passing)* menggunakan suatu fasilitas media komunikasi seperti LAN dan WAN menggunakan protokol standar seperti TCP/IP. Sistem terdistribusi dapat dipandang sebagai sebuah graf komunikasi G = (V,E), dimana V adalah himpunan prosessor dan E adalah fasilitas media komunikasi yang menghubungkan beberapa prosessor, seperti yang ditunjukkan pada **Gambar 2.1**. Karena saling berkomunikasi, kumpulan prosesor tersebut mampu saling berbagi beban kerja, data, serta sumber daya lainnya.



Gambar 2.1. *Sistem Terdistribusi*

Terdapat sekurangnya dua model dalam sistem terdistribusi ini. Pertama, sistem *client/server* yang membagi jaringan berdasarkan pemberi dan penerima jasa layanan. Pada sebuah jaringan akan didapatkan: *file server*, *time server*, *directory server*, *printer*

server, dan seterusnya. Kedua, sistem *point to point* dimana sistem dapat sekaligus berfungsi sebagai *client* maupun *server*.



Gambar 2.2. *Model Sistem Terdistribusi*

Sistem terdistribusi digunakan untuk beberapa alasan.

1. Menyelesaikan sebuah atau beberapa pekerjaan yang dapat dikerjakan secara bersama sehingga dapat diselesaikan lebih cepat dibanding jika dikerjakan dengan proses tunggal.
2. Mendistribusi dan membagi sumber daya pada beberapa lokasi agar digunakan secara bersama seperti pada printer, *shared* data, atau memori, dan lain-lain.
3. Untuk memudahkan para *user* saling bertransfer data, dan lainnya.

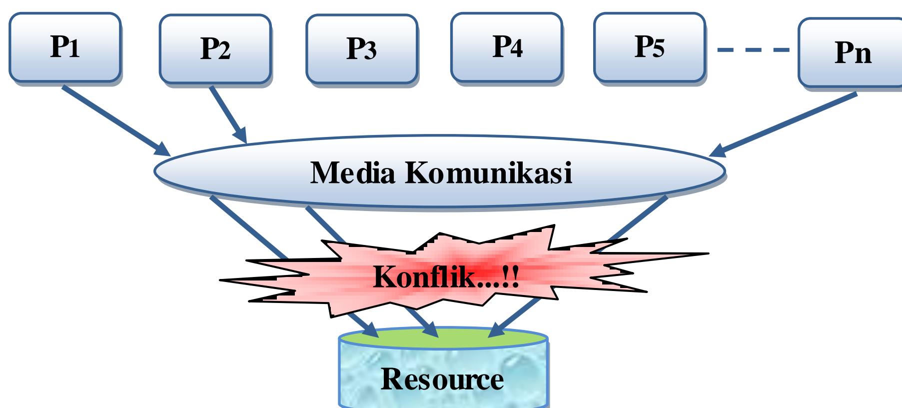
Contoh penerapan *Distributed System*: *Small Area Network (SAN)*, *Local Area Network (LAN)*, *Metropolitan Area Network* (MAN), *Online Service (OL)/Outernet*, *Wide Area Network* (WAN)/*Internet*, *Shared memory system*.

Sistem terdistribusi menyediakan akses pengguna ke berbagai sumber daya sistem. Akses tersebut menyebabkan kecepatan komputasi dan kemampuan penyediaan

data meningkat. Pada sistem terdistribusi ini terdapat beberapa *shared resource* yang digunakan secara bersama oleh komputer atau proses yang bekerja pada sistem tersebut. Dimana *resource* tersebut seringkali hanya dapat digunakan oleh sebuah proses pada satu kesatuan waktu. Akses proses pada sebuah *shared resource* yang seperti ini disebut akses *mutually exclusive*. Akses inilah yang menyebabkan terjadinya masalah *mutual exclusion.*

II.2. *Mutual Exclusion (Mutex)*

Masalah *mutual exclusion* adalah masalah singkronisasi (penyelarasan) akses ke sebuah *shared resource* tunggal dimana terdapat beberapa proses yang bersaing secara bersamaan untuk mengakses *resource* tersebut, seperti yang ditunjukkan pada **Gambar 2.3**. *Mutual exclusion* menjamin jika ada sebuah proses yang mengakses atau menggunakan variabel yang sama (digunakan juga oleh proses lain), maka proses lain akan dikeluarkan. Jadi, akses *mutual exclusive* terjadi ketika hanya ada satu proses yang boleh memakai sumber daya *(resource)*, dan proses lain yang ingin memakai sumber daya tersebut harus menunggu hingga sumber daya tadi dilepaskan atau tidak ada proses yang memakai sumber daya tersebut (misalnya : pada printer, disk drive, aplikasi data tabungan). Kondisi demikian disebut sumber daya kritis, dan bagian program yang menggunakan sumber daya kritis disebut *critical region / section*.

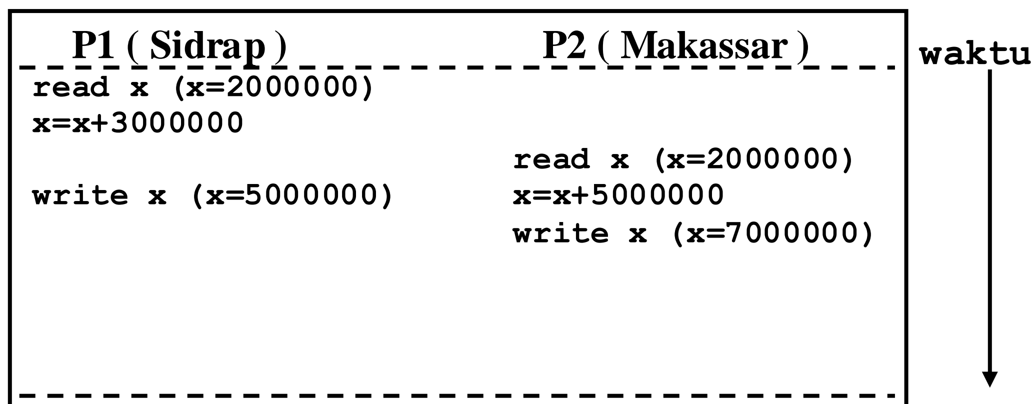


__Gambar 2.3.__ Terjadinya konflik akses beberapa proses.

Contoh:

Pada proses update data aplikasi tabungan, misalnya rekening A berisi saldo sebesar Rp.2.000.000,- terdaftar di kantor cabang di Sidrap. Pada suatu waktu, program aplikasi dikantor cabang di Makassar melayani penyetoran sebesar Rp.5.000.000,- ke rekening A tersebut. Program aplikasi membaca saldo akhir. Pada waktu bersamaan, di kantor cabang Sidrap juga melayani penyetoran sebesar Rp.3.000.000,- ke rekening A. Program aplikasi membaca saldo di kantor cabang Sidrap masih Rp.2.000.000,-. Skenario yang dapat terjadi jika *mutual exclusion* tidak dijamin yaitu:

- Program aplikasi di Sidrap dilakukan secara cepat menulis ke rekening A sehingga menghasilkan Rp.5.000.000,- Kemudian program aplikasi di Makassar menimpa hasil tersebut dengan Rp.7.000.000,-. Saldo akhir adalah Rp.7.000.000,-. Seharusnya di rekening A adalah Rp.10.000.000,-.
- Program aplikasi di Makassar dilakukan secara cepat menulis ke rekening A sehingga menghasilkan Rp.7.000.000,- Kemudian program aplikasi di Sidrap menimpa hasil tersebut dengan Rp.5.000.000,-. Saldo akhir adalah Rp.5.000.000,-. Seharusnya di rekening A adalah Rp.10.000.000,-.



Gambar 2.4. *Skenario Mutual Exclusion*

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. Pemograman harus menspesifikasikan bagian-bagian *critical section*, sehingga sistem operasi akan menjaganya.

Permasalahan *critical section* digunakan untuk mendesain sebuah protokol dimana proses-proses dapat bekerja sama. Masing-masing proses harus meminta ijin untuk memasuki *critical section*-nya. Daerah kode yang mengimplementasikan perintah ini disebut daerah *entry*. *Critical section* biasanya diikuti oleh daerah *exit*. Kode pengingat terletak di daerah *remainder*. Secara umum, solusi dari permasalahan *critical section* harus memenuhi 3 syarat yaitu :

1. ***Mutual Exclusion*.** Apabila suatu proses menjalankan *critical section*-nya, maka tidak ada proses lain yang dapat menjalankan *critical section*.
2. ***Progress (tingkat kemajuan)*.** Apabila tidak ada proses yang menjalankan *critical section*-nya dan terdapat beberapa proses yang akan memasuki *critical section*-nya, maka hanya proses-proses itu yang dapat ikut berpartisipasi di dalam keputusan proses mana yang akan memasuki *critical section* selanjutnya, dan pemilihan ini tidak dapat ditunda tiba-tiba.

3. ***Bounded Waiting (batasan menunggu).*** Ada batasan jumlah waktu yang diijinkan oleh proses lain untuk memasuki *critical section* setelah sebuah proses membuat permintaan untuk memasuki *critical section*-nya dan sebelum permintaan dikabulkan.

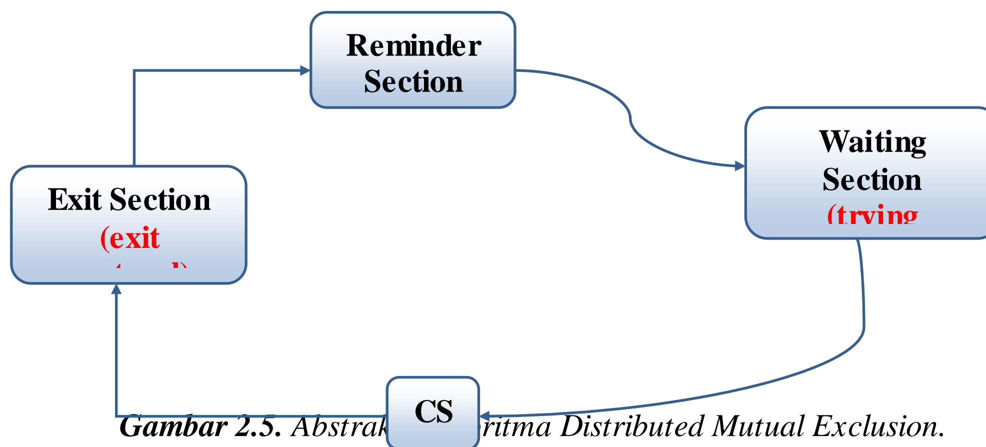
Kesuksesan proses-proses kongkurensi memerlukan pendefinisian *critical section* dan memaksakan *mutual exclusion* di antara proses-proses kongkuren yang sedang berjalan. Pemaksaan *mutual exclusion* merupakan landasan pemrosesan kongkuren. Dalam penyelesaian *mutual exclusion* harus memenuhi kriteria sebagai berikut.

1. *Mutual exclusion* harus dijamin, bahwa tidak ada proses lain, kecuali dirinya sendiri. Di sini terjadi proses tunggal.
2. Proses yang berada di *noncritical section*, dilarang mem-*blocked* proses-proses lain yang ingin masuk *critical section*.
3. Harus dijamin bahwa proses yang ingin masuk *critical section* tidak menunggu selama waktu yang tak terhingga. Jika tidak demikian maka bisa mengakibatkan masalah *lockout* dan *startvation*. *Lockout* adalah keadaan dimana terdapat proses yang telah masuk ke dalam *critical section* akan mengeksekusi *resource* berulangkali. Sedangkan *starvation* adalah penundaan masuknya sebuah proses yang telah di-*request*.
4. Ketika tidak ada proses pada *critical section*, maka proses yang ingin masuk *critical section* harus ijinkan masuk tanpa waktu tunda.
5. Tidak ada asumsi mengenai kecepatan relatif proses atau jumlah yang ada.
6. Proses hanya tinggal pada *critical section* selama satu waktu yang berhingga.

Kriteria 1(satu) merupakan pokok yang harus dipenuhi. Algoritma yang melanggar kriteria tersebut sama sekali tidak dapat digunakan. Pelanggaran kriteria -kriteria lain berarti algoritma masih dapat digunkan pada situasi tertentu tetapi harus dilakukan secara hati-hati.

II.3. Algoritma *Distributed Mutual Exclusion*

Algoritma *distributed mutual exclusion* dirancang dengan beberapa *state* berikut.



Gambar 2.5. *Abstrak*[illegible]*ritma Distributed Mutual Exclusion.*

Proses yang ingin menggunakan sumber daya *(resource)* disebut berada pada fase *reminder/Non Critical Section (NSC)*. Proses yang ingin menggunakan *resource* maka proses tersebut akan mengeksekusi *trying protocol* untuk mengirimkan *request* dan memasuki fase *Waiting Section*. Setelah *request* tersebut dibalas dan proses diberi hak menggunakan *resource*, maka proses berada pada fase *critical section*. Setelah proses selesai menggunakan *resource* maka proses kembali mengirim *request* untuk mengakhiri *critical section*-nya dengan mengeksekusi suatu *exit protocol* dan masuk pada fase *exit section*. Setelah itu kembali lagi pada fase *non critical section*, demikian seterusnya. Abstraksi algoritma ini dapat dipandang sebagai sebuah sirkulasi *state* tanpa henti seperti yang ditunjukkan pada ***Gambar 2.5***. tersebut.

Algoritma *distributed mutual exclusion* adalah mekanisme mensinkronkan prosesor dan mengakses koordinat untuk *critical section* sehingga 3 (tiga) sifat berikut terpenuhi setiap waktu.

1. ***Safety mutex***: paling banyak satu prosesor memiliki izin untuk mengeksekusi *critical section.*
2. ***Liveness*:** semua permintaan untuk *critical section* akhirnya sukses dikabulkan.
3. ***Fairness***: *critical section* dieksekusi oleh proses yang berbeda sesuai dengan urutan *request* yang dibuat.

Pada bagian ini akan dipakai algoritma-algoritma yang diaplikasikan terhadap dua proses atau lebih pada satu waktu. Misalakan dua prosesor ini adalah prosesor 1 dan 2. Dalam menghitung *mod* 2 maka 2 diindentifikasikan sebagai 0. Maka prosesor 1 dan 2 dapat ditulis sebagai proses 0 dan 1. Jadi proses tersebut diberi nama P_0 dan P_1. Untuk jelasnya, ketika menyatakan P_i, kita gunakan P_j untuk menyatakan proses yang lain, dimana $j=1-i$. Ada beberapa algoritma yang memungkinkan dapat menyelesaikan masalah *distributed mutual exclusion,* diantaranya yaitu:

II.3.1 Algoritma I

Pendekatan pertama adalah memperbolehkan semua proses menggunakan variable integer `turn` diinisialisasi ke 0 (atau 1).

```
int turn;
```

Apabila `turn = i`, maka proses P_i diijinkan untuk menjalankan *critical section* – nya. Struktur dari proses P_i adalah sebagai berikut.

```
do {
      while (turn != i) ;
            critical section
      turn = j;
            reminder section
} while (1);
```

Solusi ini sudah menjamin adanya *mutual exclusion* , karena pada suatu saat hanya ada satu proses yang masuk critical section. Pada algoritma ini masalah muncul ketika ada proses yang mendapat giliran memasuki *critical section* tetapi tidak menggunakan gilirannya sementara proses yang lain ingin mengakses *critical section*.

Misalkan ketika `turn = 1` dan P_j tidak menggunakan gilirannya maka `turn` tidak berubah dan tetap 1. Kemudian P_i ingin menggunakan *critical section*, maka ia harus menunggu sampai P_j menggunakan *critical section* dan mengubah `turn` menjadi 0. Kondisi ini tidak memenuhi syarat *progress* karena P_i tidak dapat memasuki *critical section* padahal saat itu tidak ada yang menggunakan *critical section* dan ia harus menunggu P_j mengeksekusi *non- critical section* –nya sampai kembali memasuki *critical section*.

Kondisi ini juga tidak memenuhi syarat *bounded waiting* karena jika pada gilirannya P_j mengakses *critical section* tetapi P_j selesai mengeksekusi semua kode, maka tidak ada jaminan P_i dapat mengakses *critical section* dan P_i pun harus menunggu selamanya.

II.3.2 Algoritma II

Kelemahan dengan algoritma I adalah tidak adanya informasi yang cukup tentang *state* dari masing-masing proses. Untuk mengatasi masalah ini dilakukan penggantian

variabel `turn` dengan variabel `flag`. Variabel `flag` menyimpan kondisi proses mana yang boleh masuk *critical section*. Proses yang membutuhkan akses ke *critical section* akan memberikan nilai `flag`-nya `true`. Sedangkan proses yang tidak membutuhkan *critical section* akan men-*set* nilai `flag`-nya bernilai `false`.

```
boolean flag[2];
```

Inisialisasi awal `flag [0] = flag [1] = false`. Apabila `flag[i]` bernilai `true`, nilai ini menandakan bahwa P_i siap untuk memasuki *critical section*. Struktur dari proses P_i adalah sebagai berikut.

```
do {
     flag[i] := true;
     while (flag[j]) ;
          critical section
     flag [i] = false;
          remainder section
} while (1);
```

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

Pemecahan ini menjamin *mutual exclusion*, tetapi pada algoritma ini masalah muncul ketika kedua proses secara bersamaan menginginkan *critical section*, kedua

proses tersebut akan men-*set* masing-masing `flag`-nya menjadi `true`. P_i men-*set* `flag[0] = true`, P_j men-*set* `flag[1] = true`. Kemudian P_i akan mengecek apakah P_j membutuhkan *critical section*. P_i akan melihat bahwa `flag[1] = true`, maka P_i akan menunggu sampai P_j selesai menggunakan *critical section*. Namun pada saat bersamaan, P_j juga akan mengecek apakah P_i membutuhkan *critical section* atau tidak, ia akan melihat bahwa `flag[0] = true`, maka P_j juga akan menunggu P_i selesai menggunakan *critical section*-nya.

Kondisi ini menyebabkan kedua proses yang membutuhkan *critical section* tersebut akan saling menunggu dan "saling mempersilahkan" proses lain untuk mengakses *critical section*, akibatnya malah tidak ada yang mengakses *critical section*. Kondisi ini menunjukkan bahwa Algoritma II tidak memenuhi syarat *progress* dan syarat *bounded waiting*, karena kondisi ini akan terus bertahan dan kedua proses harus menunggu selamanya untuk dapat mengakses *critical section*.

II.3.3 Algoritma III (Peterson)

Algoritma III ditemukan oleh G.L. Petterson pada tahun 1981 dan dikenal juga sebagai Algoritma Petterson. Petterson menemukan cara yang sederhana untuk mengatur proses agar memenuhi *mutual exclusion*. Algoritma ini adalah solusi untuk memecahkan masalah *critical section* pada dua proses. Ide dari algoritma ini adalah menggabungkan variabel yang di-*sharing* pada Algoritma I dan Algoritma II, yaitu variabel `turn` dan variabel `flag`. Sama seperti pada Algoritma I dan II, variabel `turn` menunjukkan giliran proses mana yang diperbolehkan memasuki *critical section* dan

variabel `flag` menunjukkan apakah suatu proses membutuhkan akses ke *critical section* atau tidak.

```
int turn;
boolean flag[2];
```

Inisialisasi `flagI[0] = flag[1] = false` dan nilai dari `turn` bernilai 0 atau 1. Kemudian jika suatu proses ingin memasuki *critical section*, ia akan mengubah `flag`-nya menjadi `true` (memberikan tanda bahwa ia butuh *critical section*) lalu proses tersebut memberikan `turn` kepada lawannya. Jika lawannya tidak menginginkan *critical section* (`flag`-nya `false`), maka proses tersebut dapat menggunakan *critical section*, dan setelah selesai menggunakan *critical section* ia akan mengubah `flag`-nya menjadi `false`. Tetapi apabila proses lawannya juga menginginkan *critical section* maka proses lawan-lah yang dapat memasuki *critical section*, dan proses tersebut harus menunggu sampai proses lawan menyelesaikan *critical section* dan mengubah `flag`-nya menjadi `false`. Struktur dari proses P_i adalah:

```
do {
      flag [i]:= true;
      turn = j;
      while (flag [j] and turn = j) ;
          critical section
      flag [i] = false;
          remainder section
} while (1);
```

Jadi, algoritma III (Peterson) ini telah memenuhi ketiga syarat diatas yaitu *mutual exclusion*, *progress* dan *bounded waiting* dan dapat digunakan untuk memecahkan permasalahan *critical section* untuk dua proses.

II.3.4 Algoritma Dijkstra Semaphore

Semaphore adalah salah satu cara menangani *critical section*, yang dikembagkan oleh *Edsgitiner Dijkstra*. Pada algoritma Dijkstra mengalami kekurangan yaitu adanya *busy waiting*. Apabila suatu proses menempati *critical section,* maka akan terjadi iterasi secara terus -menerus pada entry section. Hal ini akan menimbulkan masalah pada sistem yang menggunakan konsep *multiprogramming* seperti pada masalah *mutual exclusion* ini.

Prinsip *semaphore* adalah dua proses atau lebih dapat bekerja sama dengan menggunakan 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 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 P_1 dengan pernyataan S_1 dan P_2 dengan pernyataan S_2. Andaikan kita mengharapkan S_2 baru akan

dijalankan hanya setelah S_1 selesai. Hal ini dapat dilakukan dengan menggunakan bantuan *semaphore synch* (dengan nilai awal =0) yang akan di-*share* oleh kedua proses.

Untuk Proses P_1 :

S_1;

signal(synch);

Untuk proses P_2 :

wait(synch);

S_2;

Karena nilai awal untuk synch adalah nol, maka P_2 akan mengeksekusi S_2 hanya setelah P_1 mengerjakan signal (synch) setelah S_1.

Jadi, 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*.

II.4. PROMELA dan SPIN

PROMELA merupakan sebuah bahasa yang dikembangkan oleh Gerrard Holzmann untuk memodelkan sistem, khususnya protokol komputer. PROMELA, yang merupakan kependekan dari PROcess MEta LAnguage, menggunakan basis CSP, bahasa C, dan SDL (*Specification Description Language*).

Selain mengembangkan bahasanya, Holzmann juga mengembangkan sebuah tools yang diberi nama SPIN untuk melakukan validasi, tepatnya *model checking* terhadap

spesifikasi yang ditulis dalam bahasa PROMELA. Bahkan, sebetulnya buku dan makalah dari Holzmann lebih banyak difokuskan kepada algoritma-algoritma yang dituangkan dalam bentuk tools SPIN tersebut.

II.4.1 PROMELA

Promela adalah bahasa pemodelan verifikasi. Ini menyediakan sarana untuk membuat abstraksi protokol (atau sistem terdistribusi secara umum) yang menekan detail yang berhubungan dengan proses interaksi. Secara singkat, di dalam bahasa PROMELA ada tiga buah *objects* yang berbeda, yaitu *process, message channels,* dan *variabel* (global atau lokal). Semua proses dalam PROMELA adalah obyek global, sedangkan *variable* dan *channel* dapat dideklarasikan dalam suatu proses.

Deskripsi dalam bahasa PROMELA ditulis dalam bentuk *statement*. Dalam PROMELA tidak ada perbedaan antara *conditions* dan *statements*. Eksekusi dari *statement* ini bergantung kepada sifat *executability*-nya. Statement ini bisa dijalankan (*executable*) atau terhalang (*blocked*) bergantung kepada nilai dari *variable* atau isi dari *message channel.* Jika kondisinya benar (*holds*) maka *statement* akan dieksekusi. Jika tidak, maka *statement* akan terhenti (*blocked*) sampai kondisi menjadi benar.

Ada enam (6) jenis data yang sudah terdefinisi di PROMELA yaitu:

- *bit* (ukuran implementasi 1 bit)
- *bool* (1 bit)
- *byte* (8 bit)
- *short* (16 bit)

- *chan* (bergantung kepada data). Jenis data *chan* ini digunakan untuk mengimplementasikan *message channel*, yaitu sebuah obyek yang dapat menyimpan beberapa nilai yang dijadikan satu.

Perlu diketahui bahwa *declaration* dan *assignment* selalu dapat dieksekusi, jadi selalu *executable*. Tanda titik koma (; atau *semi-colon*) merupakan tanda pemisah statement atau *statement separator*. Dia bukan *statement terminator* sehingga pada statement yang terakhir tidak perlu diakhiri dengan tanda titik koma ini. Ada dua *statement separator* dalam PROMELA, yaitu titik koma dan tanda panah - >.

Proses di dalam PROMELA didefinisikan dengan menggunakan keyword *proctype*, seperti contoh berikut.

```
proctype A() { byte state; state = 3}
```

Contoh di atas menunjukkan definisi dari proses A, dimana di dalamnya ada sebuah variabel lokal yang bernama *state*. Kemudian variabel ini kita masukkan nilai “3”.

PROMELA memiliki tiga *control flow*; *case selection*, *repetition*, dan *unconditional jump*. Pemilihan atau *case selection* dilakukan dengan menggunakan kata kunci *if if* dan tanda titik dua (:) dua kali.

```
if
:: (a != b) -> option1
:: (a == b) -> option2
fi
```

Dalam contoh di atas, *statement* pertama berfungsi sebagai *guard*, yang ikut menentukan sifat *executable* dari *statement* tersebut. Dalam satu saat dipilih salah satu dari pilihan tersebut dengan syarat bahwa statement tersebut *executable*. Dalam contoh

di atas, hanya salah satu *statement* akan dipilih bergantung kepada *guard* yang ada, yaitu bergantung kepada nilai variabel “a” dan “b”. Namun dalam contoh lain bisa terjadi bahwa ada lebih dari satu pilihan yang bisa dipilih. Untuk kasus ini akan dipilih salah satu secara random.

```
if
:: (x > y) -> maks = x
:: (y > x) -> maks = y
:: (x == y) -> maks = x
:: (x != y) -> flag = 1
fi
```

Dalam contoh di atas jika $x = 5$ dan $y = 7$ maka ada dua pilihan yang dapat dipilih, yaitu pilihan kedua dengan guard $(y > x)$ atau pilihan keempat dengan *guard* $(x! = y)$. Pilihan akan dipilih secara random. Jika semua pilihan tidak ada yang *executable*, maka proses akan terhambat (*blocked*) sampai ada salah satu yang bisa dieksekusi. Pengulangan (*repetition*) dilakukan dengan keyword *do od*.

```
byte count;
proctype counter()
{
do
:: count = count +1
:: count = count -1
:: (count == 0) -> break
od
}
```

Di dalam PROMELA diperkenankan untuk melakukan *unconditional jump* dengan kata kunci *goto*. *Flow* akan diteruskan kepada bagian (label) yang dituju oleh *goto*. Dalam contoh di bawah ini apa bila nilai *x* sama dengan *y* maka *flow* akan

diteruskan ke label “done” yang kemudian dilanjutkan dengan perintah *skip* yang tidak melakukan apa-apa.

```
proctype Euclid(in x, y)
{
do
:: (x > y) -> x = x y
:: (x < y) -> y = y x
:: (x == y) -> goto done
od
     done:
     skip
}
```

II.4.2 SPIN

SPIN (Simple Promela INterpreter) adalah paket software yang dikembangkan di Grup Spesifikasi dan Verifikasi Formal Laboratorium Bell. SPIN dapat digunakan untuk menelusuri kesalahan-kesalahan logika pada desain sistem terdistribusi, melakukan *model checking* terhadap sistem yang dijelaskan dalam bahasa pemodelan yang disebut PROMELA (PROses MEta LAnguage), bahasa yang memungkinkan untuk menciptakan proses dinamis bersamaan.

SPIN ditulis dalam bahasa C dan tersedia *source code*-nya untuk sistem UNIX dan Windows. Ada juga *front end* grafis berbasis Tcl/Tk. Pengguna tidak perlu melihat kode C, walaupun untuk menggunakan Spin kita harus memiliki sebuah program kompilasi C.

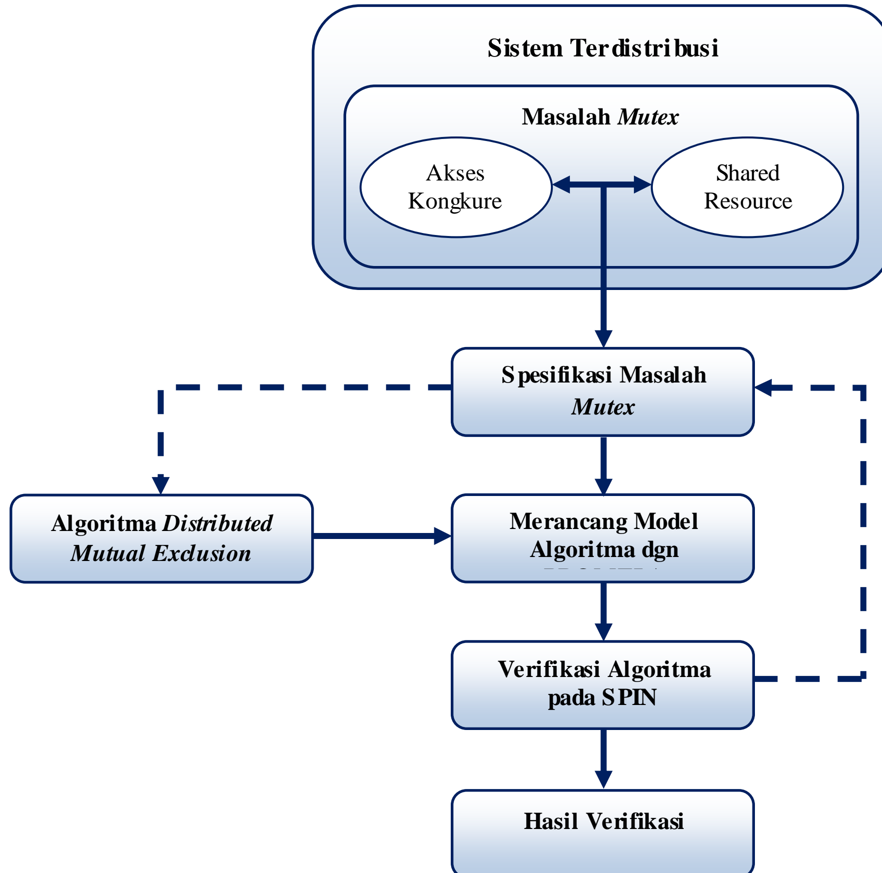
Tool ini melakukan pemeriksaan terhadap inkonsistensi dalam suatu spesifikasi, mencari resepsi yang tidak lengkap, ketidaklengkapan *flags*, dsb. SPIN melakukan

analisa terhadap suatu spesifikasi dalam 3 langkah, yaitu melakukan simulasi, melakukan pencarian ekshaustif, dan melakukan *bit space partial order reduction* untuk spesifikasi yang besar.

Verifikasi di Spin dapat dijalankan dengan tiga-line (menghasilkan verifikasi, mengkompilasi, dan menjalankannya). Dalam Spin terdapat 3 (tiga) mode yang biasa digunakan, yaitu:

- *Random Simulation Mode* (modus simulasi random), menggunakan nomor acak generator untuk menyelesaikan nondeterminan yang berada dalam program kongkuren (dari proses yang akan dieksekusi). Dalam mode ini, Spin bisa menggantikan *concurrency* klasik simulator sebagai *tools* untuk mempelajari program kongkuren atau bersamaan
- *Interactive Simulation Mode* (modus simulasi interaktif), memungkinkan pengguna untuk memilih instruksi berikutnya yang akan dieksekusi. *Interactive Simulation* juga didukung *simulator concurrency* yang penting untuk menunjukkan skenario seperti *starvation* atau *fairness* yang tidak mungkin terjadi secara acak.
- *Verification Mode* (modus verifikasi), sistematis mencari balik seluruh *entire state* yang melanggar perhitungan.

II.5. Alur Kerja



Gambar 2.6 *Alur Kerja*

BAB III

ALGORITMA DISTRIBUTED MUTUAL EXCLUSION

Dari beberapa algoritma yang telah diuraikan pada bab II sebelumnya, maka algoritma yang memenuhi syarat untuk dapat kita gunakan dalam menyelesaikan masalah *distributed mutual exclusion* hanya algoritma Peterson dan algoritma Dijkstra Semaphore.

Dalam bahasan bab ini akan dibuktikan kedua algoritma tersebut apakah juga benar-benar telah memenuhi semua sifat dalam menyelesaikan masalah *distributed mutual exclusion.* Untuk itu, sebelumnya akan dibuktikan *lemma* yang akan menghasilkan *teorema* yang menjamin sifat-sifat *mutual exclusion* tersebut.

III.1. Algoritma Peterson

Kali ini prosesor pada algoritma Peterson dilambangkan dengan indeks i dimana $i \in \{0,1\}$, dan kemudian j untuk menyatakan indeks prosesor lain (1 - i). Seperti pada bab sebelumnya juga bahwa ketika proses dinyatakan P_i, kita gunakan $P_j$untuk menyatakan proses yang lain, dimana $j = 1 - i$.

Lemma 1.

Algoritma Peterson menjamin sifat mutual exclusion; safety mutex.

Bukti. Algoritma Peterson menjamin sifat *mutual exclusion* yaitu *safety mutex* dalam artian hanya satu proses yang dapat mengeksekusi *critical section* pada suatu waktu. Misalkan P_i dan P_j ingin mengakses *critical section* secara bersamaan, kedua proses akan menset masing-masing `flag` menjadi `true` (`flag[0]=true` dan`flag[1]=true`), dalam kondisi ini P_i dapat mengubah `turn=1` dan Pj juga dapat

mengubah `turn=0` dan begitu juga sebaliknya. Kondisi `turn` inilah yang mengakibatkan $P_i$dan P_j dapat mengakses *critical section*. Tetapi, kondisi `turn=1` pada P_i terjadi jika P_j adalah pesaing bagi P_i dan kondisi `turn=0` pada P_j terjadi jika P_i adalah pesaing bagi P_j. Hal ini bertentangan dengan asumsi awal bahwa $P_i$dan P_j mengakses *critical section* secara bersamaan (Lynch,, 1996).

Lemma 1 menyatakan bahwa algoritma Peterson menjamin sifat *mutual exclusion* yaitu *safety mutex*. Sedangkan *lemma* berikut ini akan dipaparkan bahwa algoritma Peterson juga menjamin sifat *liverness* dan *fairness* yaitu semua permintaan untuk *critical section* akhirnya dikabulkan dan dieksekusi oleh proses yang berbeda sesuai dengan *request* yang dibuat.

Lemma 2.

Algoritma Peterson menjamin sifat liverness dan fairness.

Bukti. Andaikan algoritma Peterson tidak menjamin sifat *liverness* dan *fairness*, artinya akan dibuktikan bahwa ada proses yang terus menerus dalam *waiting section* dan ada proses yang dapat mengakses *critical section* berulangkali. Misalkan, ketika P_i membutuhkan *critical section*, maka P_i akan mengubah `flag[0]=true`, lalu P_i mengubah `turn=1`. Jika P_j mempunyai `flag[1]=false`, dalam artian proses tidak membutuhkan akses ke *critical section*, (berapapun nilai `turn`) maka P_i-lah yang dapat mengakses *critical section*. Namun apabila sebaliknya, P_j juga membutuhkan *critical section*, karena `flag[1]=true` dan `turn=1`, maka P_j yang dapat memasuki *critical section* dan P_i harus menunggu sampai P_j menyelesaikan *critical section* dan mengubah `flag[1]=false`, setelah itu barulah P_i dapat mengakses *critical section* (Lynch, 1996).

□

□

Dengan demikian, tidak terbukti bahwa ada proses yang dapat mengakses *critical section* berulangkali dan terus menerus dalam *waiting section*-nya. Jadi, semua permintaan untuk *critical section* akhirnya dikabulkan dan dieksekusi oleh proses yang berbeda sesuai dengan *request* yang dibuat.

Teorema 1.

Algoritma Peterson menyelesaikan masalah distributed mutual exclusion; menjamin sifat safety mutex, liverness dan fairness.

Bukti. Berdasarkan dari *Lemma 1* dan *Lemma 2* maka dapat disimpulkan bahwa *teorema 1* tersebut terbukti. □

III.2. Algoritma Dijkstra Semaphore

Untuk algoritma Dijkstra Semaphore, variabel `turn` merupakan suatu bilangan bulat $\{1....,n\}$ dan variabel `flag[i]`, $1 \leq i \leq n$, artinya satu setiap proses. Jadi dimisalkan masing-masing mengambil pada nilai dari $\{0,1\}$, nilai awal 0. Variabel `turn` di sini dapat dipakai dan terbaca oleh semua prosesor. Masing-masing `flag[i]` dipakai hanya oleh prosesor *i* tetapi pembacaannya oleh semua prosesor.

Teorema 2.

Algoritma Dijkstra Semaphore menjamin sifat mutual exclusion; safety mutex.

Bukti. Algoritma Dijkstra Semaphore menjamin sifat *mutual exclusion* yaitu *safety mutex,* dalam artian hanya satu proses yang dapat mengeksekusi *critical section* pada suatu waktu. Kondisi ini dapat ditunjukkan dengan misalnya, pada proses ke-i stage pertama, prosesor memulai dengan mengatur `flag`-nya sama dengan 1 dan kemudian berulang kali memeriksa variabel `turn` untuk melihat apakah `turn:=1`. Jika tidak ,

dan jika pemilik arus dari `turn` terlihat tidak aktif (`flag` *(*`turn`*)*=0), prosesor *i* mengatur `turn:=1`. Setelah terlihat `turn:=1`*,* prosesor *i* pindah ke stage kedua.

Pada stage kedua, prosesor *i* mengatur lagi `flag`-nya, untuk meninggalkan *critical section*, sehingga prosesor *i* menurunkan `flag`-nya kembali ke 0 dan kemudian memeriksa dengan melihat bahwa tidak ada prosesor lain yang mempunyai `flag=`0. Pemeriksaan dari `flag-flag` prosesor lain dapat dikerjakan pada setiap urutan. Jika pemeriksaan lengkap dan sukses, prosesor *i* ke *critical section*, sebaliknya, ia kembali ke stage pertama. (Lynch, 1996).

□*eorema2* telah menyatakan bahwa algoritma Dijkstra Semaphore menjamin sifat *mutual exclusion* yaitu *safety mutex*. Sedangkan *teorema* berikutnya juga akan dipaparkan apakah terbukti menjamin sifat *liverness* dan *fairness* atau tidak.

Teorema 3.

Algoritma Dijkstra Semaphore tidak menjamin sifat liverness dan fairness..

Bukti. Andaikan algoritma Dijkstra Semaphore menjamin sifat *liverness* dan *fairness*, artinya akan dibuktikan bahwa tidak ada prosesor yang terus menerus berada dalam *waiting section* dan dapat mengakses *critical section* berulangkali.

Misalkan prosesor dalam sistem yang terakhir kali mengeksekusi *critical section* adalah prosesor 2 (P_1), berarti nilai `flag`–nya adalah 0 dan nilai variabel `turn`-nya adalah 1. Kemudian prosesor lainnya yaitu (P_0) ingin mengakses *critical section*, berarti (P_0) akan men-*set* `flag`–nya=`1`. Kemudian (P_0) akan mengecek variabel `turn` apakah prosesor yang terakhir kali mengakses *critical section* (P_1) bukan 1, untuk mendapatkan hak akses ke *critical section*-nya. Karena nilai `turn` pada $(P_1) = 1$, maka

(P_0) belum bisa mengakses *critical section*-nya, jadi (P_0) dalam keadaan menunggu (*waiting)* sampai (P_1) men-*set* kembali `turn` menjadi 0. Sebaliknya jika tidak, (P_0) dapat mengakses *critical section*-nya sampai meninggalkannya kembali ke daerah *reminder*-nya dan men-*set* `flag`–nya=`0`.

Untuk kondisi di atas, (P_0) tidak dapat memasuki *critical section* padahal saat itu tidak ada yang menggunakan *critical section* dan ia harus menunggu (P_1) mengeksekusi *non-critical section*–nya sampai kembali memasuki *critical section.*

Jika (P_1) kembali men-*set* `flag`–nya=`1`, kemudian menjalankan perintah-perintah algoritma hingga mendapatkan hak akses mengeksekusi *critical section*-nya lagi, lalu kembali ke daerah *reminder*-nya. Maka tidak ada jaminan pada (P_0) mendapat hak mengakses *critical section* dan (P_0) pun harus menunggu selamanya. (Lynch, 1996). □

Dengan demikian, *teorema 3* terbukti karena memang benar ada prosesor yang dapat mengakses *critical section* berulangkali dan ada yang terus menerus dalam *waiting section*-nya.

Jadi, algoritma Dijkstra Semaphore tersebut tidak menjamin semua sifat *mutual exclusion* yaitu ada sifat *liverness* dan *fairness* yang tidak terpenuhi. Tetapi, karena algoritma yang ini telah memenuhi sifat pokok dari penyelesaian masalah *distributed mutual exclusion*yaitu *safety mutex* seperti yang telah dijelaskan pada bab II sebelumnya, sehingga masih dapat digunakan pada situasi tertentu tetapi harus dilakukan secara hati-hati.

BAB IV

HASIL DAN PEMBAHASAN

Dalam bab ini akan membahas mengenai spesifikasi rancangan algoritma *distributed mutual exclusion* yang dimodelkan dalam bahasa PROMELA *(Protocol/Process Meta Language)* dan kemudian akan diverifikasi menggunakan tools SPIN *(Simple Promela Interpreter)* untuk mengetahui apakah prinsip kerja dari algoritma tersebut sudah benar didalam mengatur penangan masalah *distributed mutual exclusion.*

IV.1. Spesifikasi Rancangan Algoritma

Spesifikasi yang dibuat dalam merancang algoritma adalah sesuai dengan prinsip kerja algoritma pada setiap penyelesaian masalah *mutual exclusion* pada bahasan bab II sebelumnya.

Adapaun rancangan dari algoritma untuk penyelesaian *distributed mutual exclusion* ini dimodelkan dalam bahasa PROMELA sebagai berikut.

IV.1.1 Algoritma Peterson

Implementasi program untuk algoritma 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) } }
```

Implementasi program untuk algoritma Dijkstra Semaphore adalah sebagai berikut.

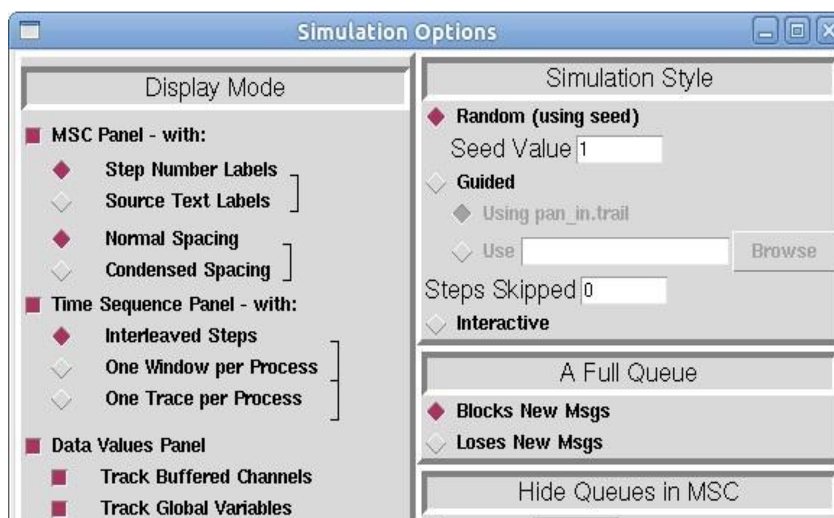
```
#define p 0
#define v 1
chan sema = [0] of { bit };
proctype dijkstra()
{ byte count = 1;
do
:: (count == 1) -> sema!p; count = 0
:: (count == 0) -> sema? v; count = 1
od
}
proctype user()
{ do
:: sema? p;
/* critical section */
sema!v;
/* non-critical section */
od
}
init
{ run dijkstra(); run user();
}
```

ributed

lengan

IV.2.1 Simulasi

Simulasi yang dilakukan menggunakan XSPIN versi-5.2.5 di atas platform Ubuntu Linux 10.10. Adapun parameter simulasinya dapat diatur seperti pada ***Gambar 4.1*** berikut ini.



Gambar 4.1 *Window Parameter Simulasi*

Pada window diatas ada beberapa hal yang dapat diatur berkaitan dengan simulasi yang akan dilakukan, seperti misalnya panel apa saja yang akan ditampilkan selain panel standar *Simulation Output*.

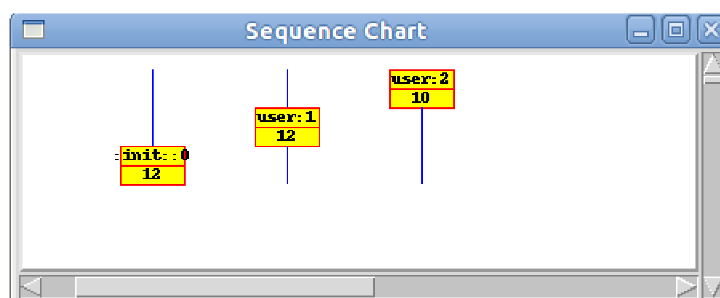
- Panel *Message Sequence Chart* (MSC) akan menampilkan representasi grafis dari interaksi (pertukaran *message*) antar proses. Gambar yang muncul secara default dapat disimpan dalam file msc.ps.
- Panel *Time Sequence* (TS) menghasilkan representasi tekstual dari informasi yang serupa dengan yang muncul 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 algoritma untuk penyelesaian masalah *distributed mutual exclusion* sebagai berikut.

IV.2.1.1 Simulasi Algoritma Peterson

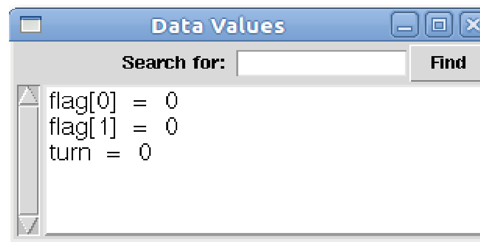
Simulasi algoritma Peterson dengan menggunakan tools SPIN sebagai berikut.

a. Snapshot panel Message Sequence Chart.



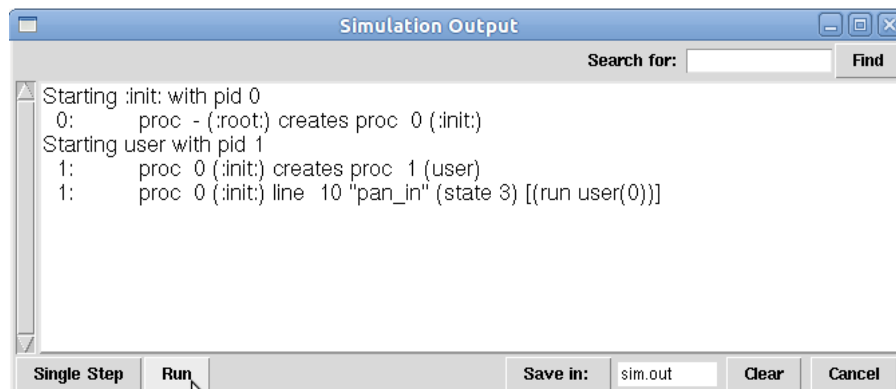
Gambar 4.2 *Tampilan akhir MSC*

b. Snapshot panel Data value :

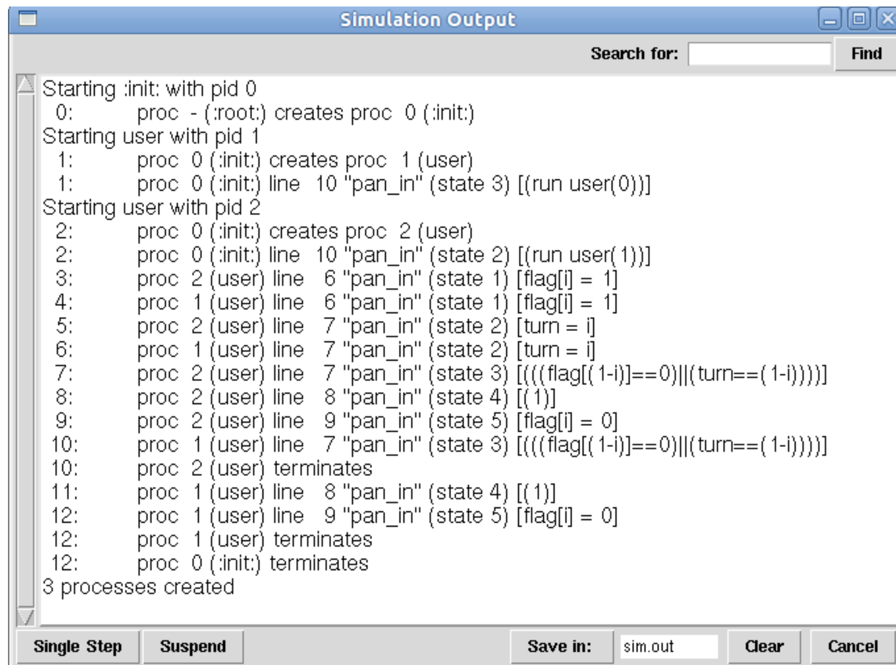


Gambar 4.3 *Data value*

c. Snapshot panel Simulation output :

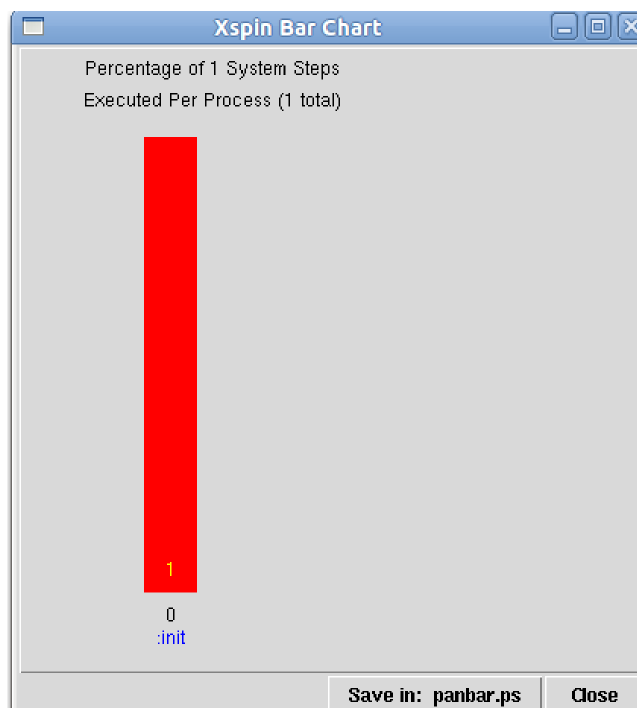


Gambar 4.4 *Awal simulasi*

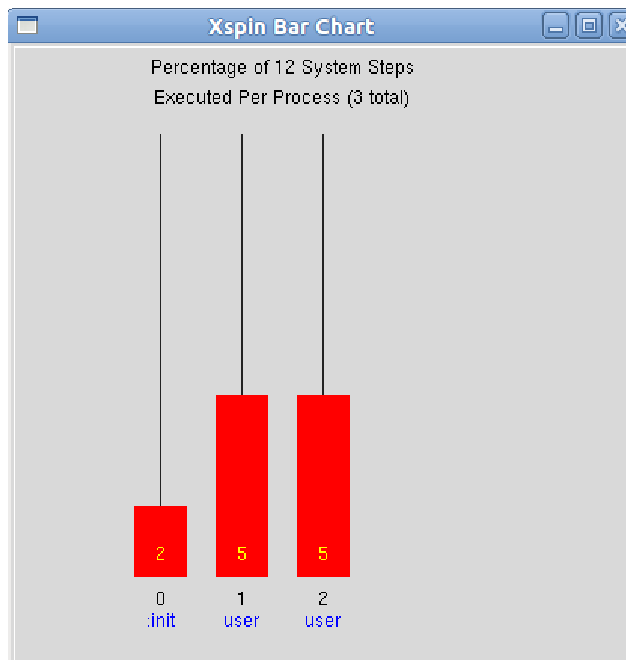


Gambar 4.5 *Akhir simulasi*

d. Snapshot panel execution bar chart :



Gambar 4.6 *Awal execution bar chart*



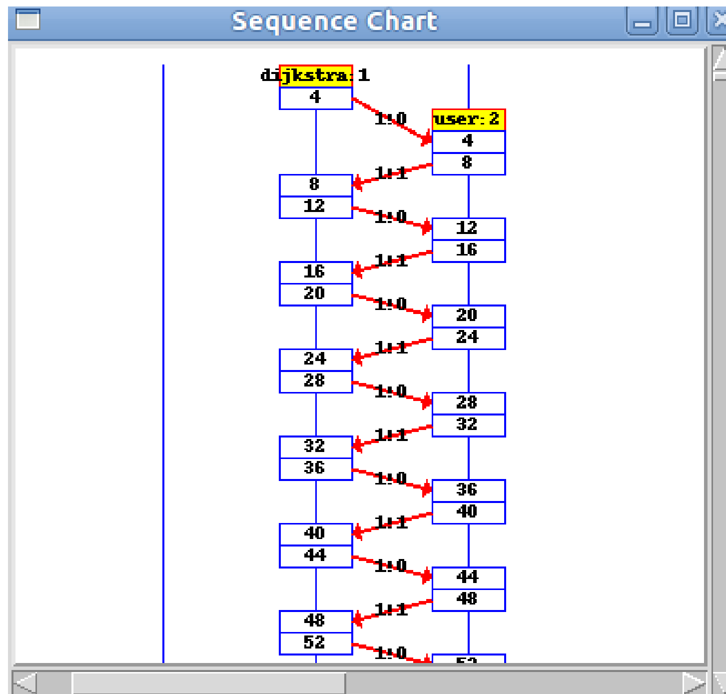
Gambar 4.7 *Akhir execution bar chart*

Dari simulasi dapat dilihat bahwa kedua proses dapat saling bergantian memasuki *critical section*. Prinsip masuk ke *critical section*, proses P_0 meng-*set* `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* begitu pula sebaliknya yang terjadi pada P_1

IV.2.1.2 Simulasi Algoritma Dijkstra Semaphore

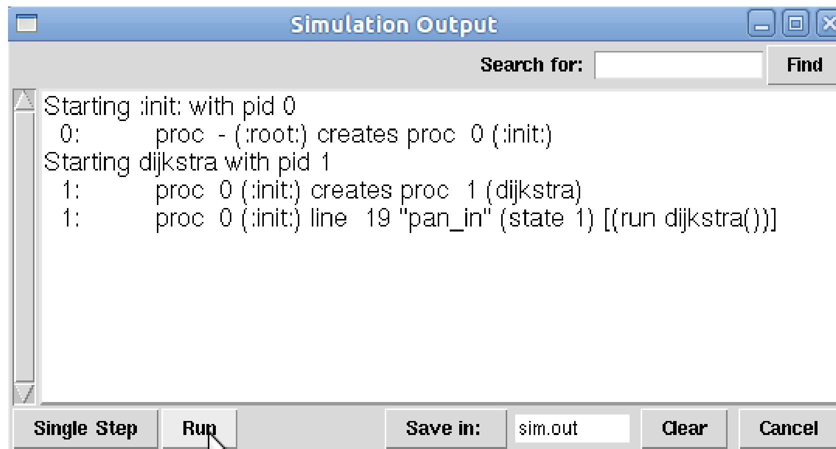
Simulasi algoritma Dijkstra Semaphore dengan menggunakan tools SPIN sebagai berikut.

1. Snapshot panel Message Sequence Chart.

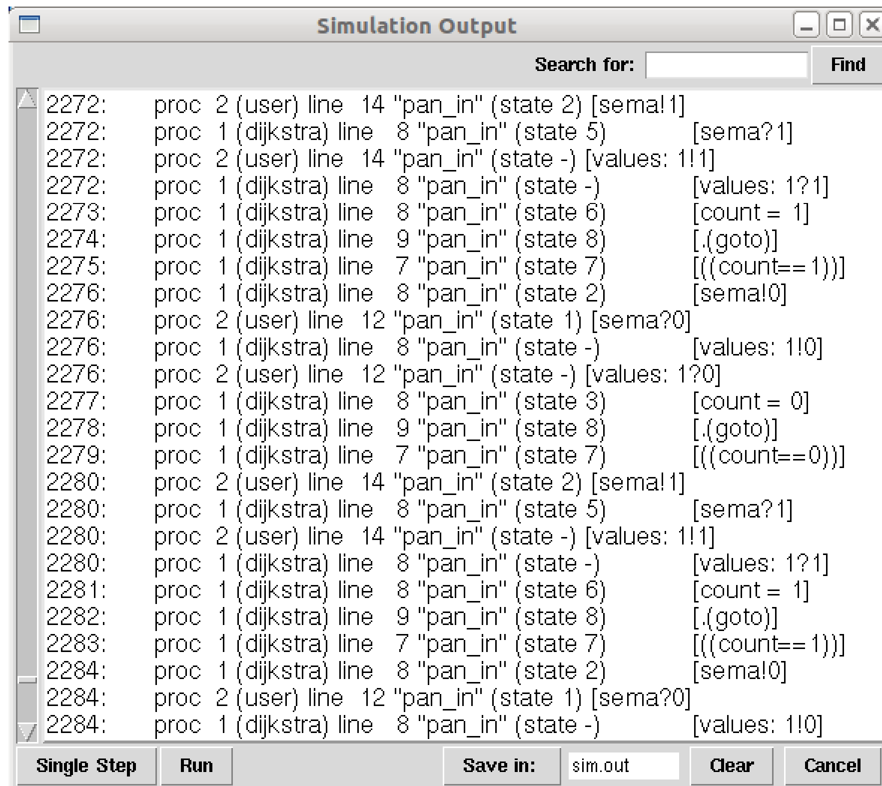


Gambar 4.8 *Sebagian Tampilan MSC*

2. Snapshot panel Simulation output :

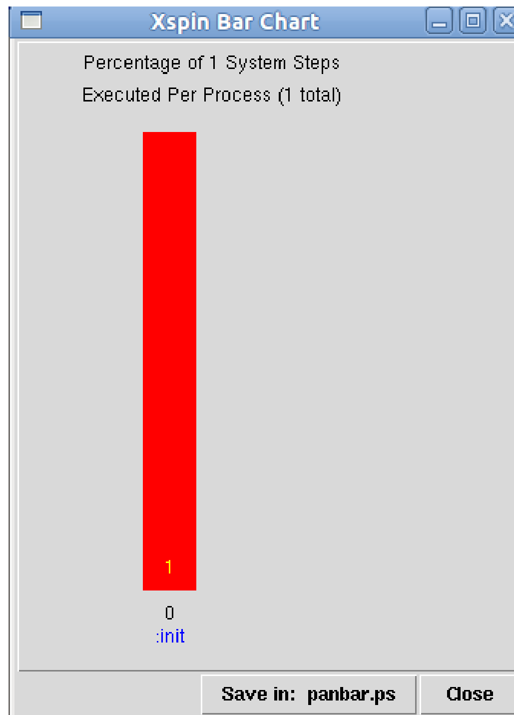


Gambar 4.9 *Awal simulasi*

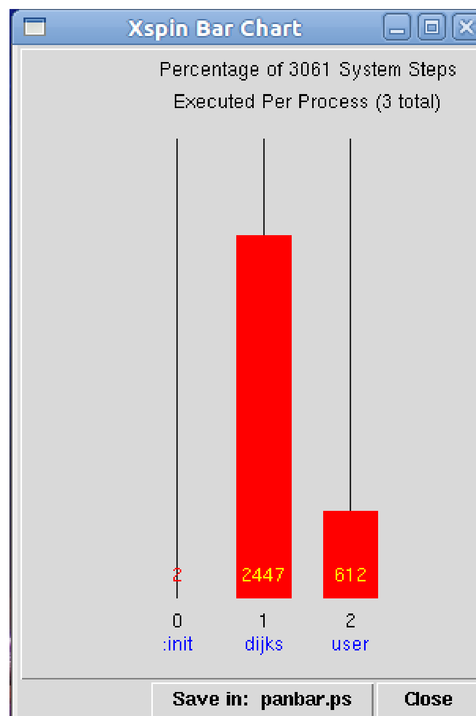


Gambar 4.10 *Sebagian simulasi*

3. Snapshot panel execution bar chart :



Gambar 4.11 *Awal execution bar chart*



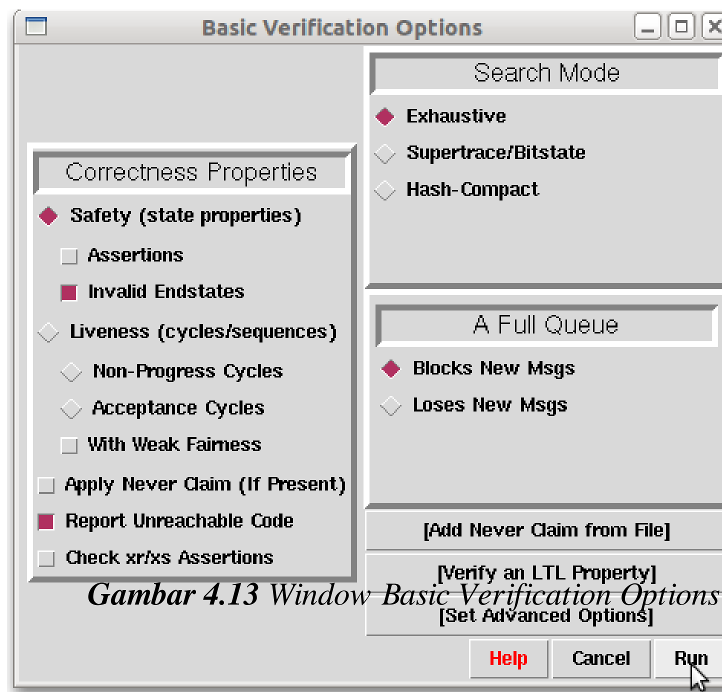
Gambar 4.12 *Sebagian execution bar chart*

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

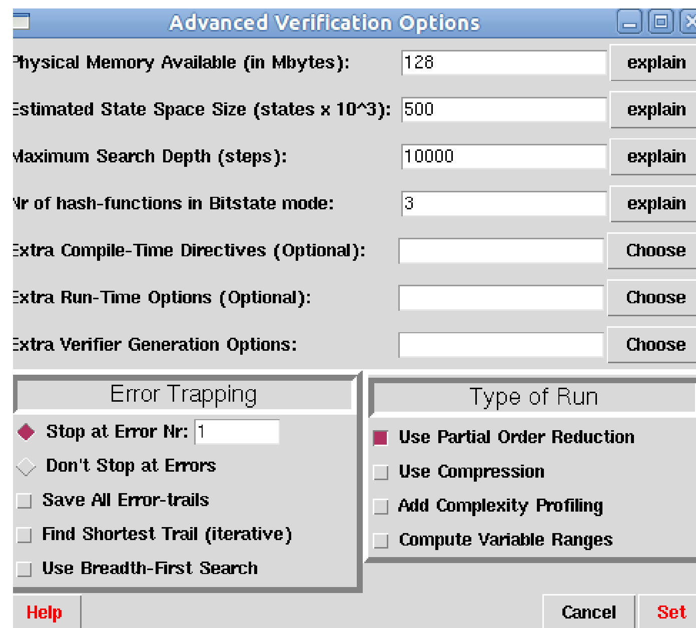
IV.2.2 Verifikasi

Keberhasilan dari simulasi yang telah dilakukan belum dapat menunjukkan baik atau tidaknya hasil kerja setiap algoritma tersebut untuk penyelesaian masalah *distributed mutual exclusion*. Untuk dapat menguji apakah algoritma tersebut telah memenuhi semua sifat dalam menyelesaikan masalah *distributed mutual exclusion,* (seperti yang telah diuraikan pada sebelumnya) terpenuhi atau tidak, maka harus dilakukan verifikasi.

Dengan tools XSPIN tadi, pengaturan parameter verifikasi dilakukan pada window *Verification Option,* seperti terlihat pada ***Gambar 4.13***. Dan masih dapat dilakukan pengaturan lanjutan pada window *Advanced Verification Option*.



Gambar 4.13 *Window Basic Verification Options*



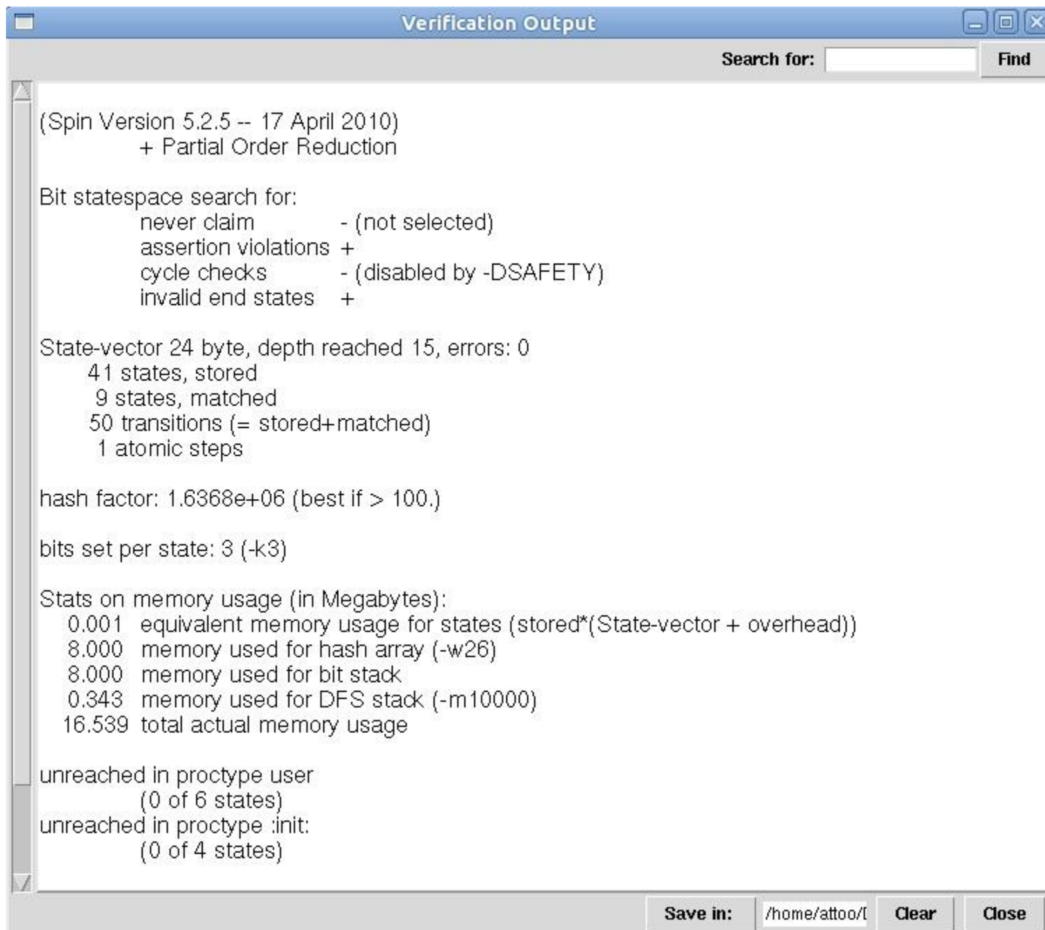
Gambar 4.14 *Window Advanced Verification Options*

Verifikasi algoritma untuk spesifikasi penyelesaian masalah *distributed mutual exclusion* yang dilakukan adalah dari kondisi berikut ini.

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

IV.2.2.1 Verifikasi Algoritma Peterson

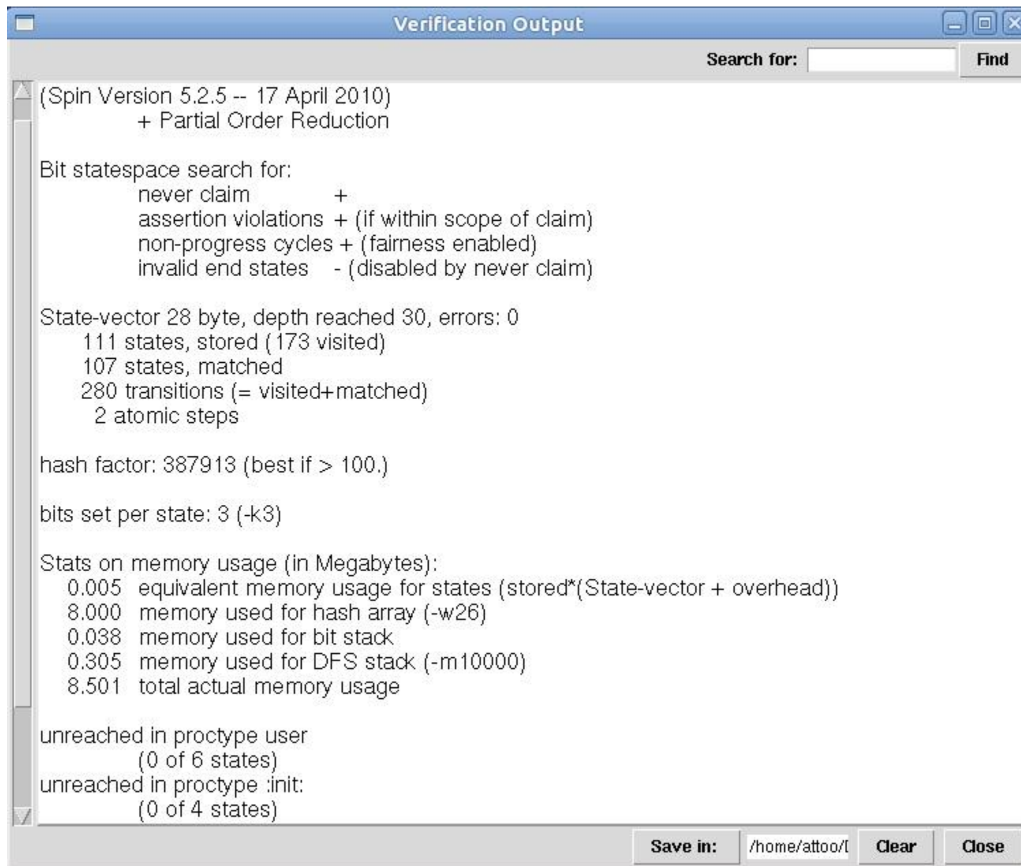
Verifikasi untuk spesifikasi pada kondisi *Safety (state properties)* dengan option *assertion* dan *invalid endstates* kita hidupkan, modus pencariannya *supertrace/bitstate*, maka didapat hasil verifikasi sebagai berikut.



Gambar 4.15 *Hasil Verifikasi untuk spesifikasi Safety Condition*

Dari hasil verifikasi ini dapat dilihat bahwa algoritma Peterson tidak mengalami masalah dalam *assertion violation* dan juga pada state yang dibangun, artinya tidak didapatkan proses yang melanggar *mutual exclusion*.

Verifikasi untuk spesifikasi pada kondisi *Liveness (cycles/seguence)* dengan option *non–progress cycles* dan *with weak fairness* kita hidupkan, modus pencariannya *supertrace/bitstate*, maka didapat hasil verifikasi sebagai berikut.

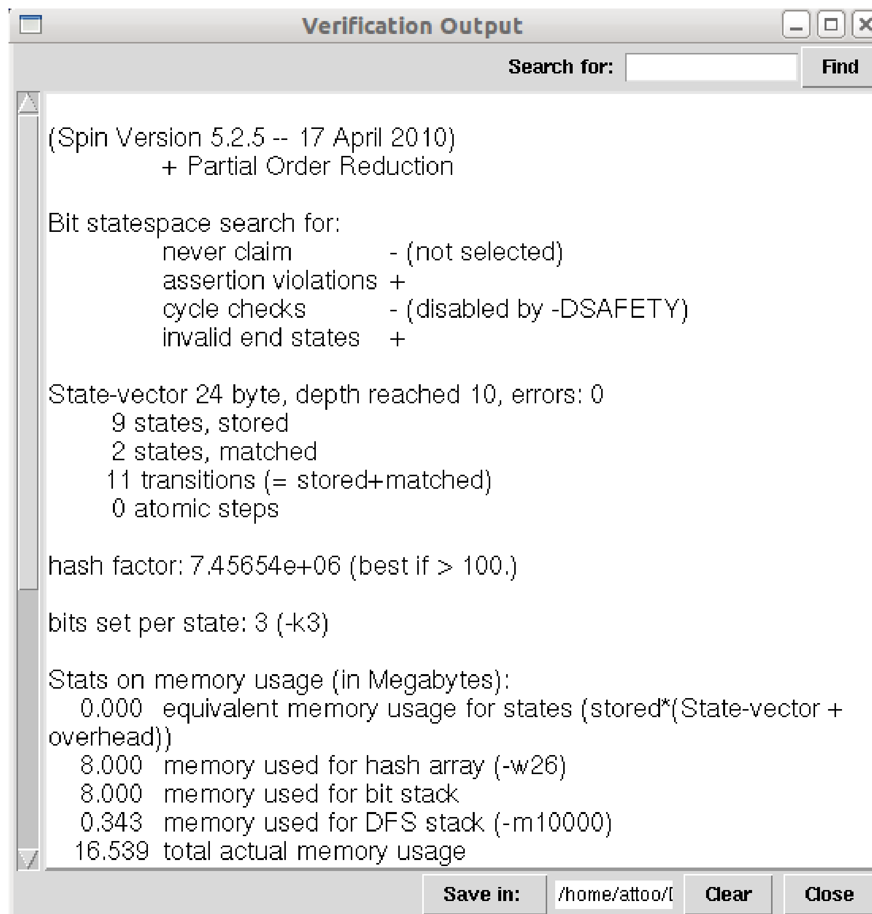


Gambar 4.16 *Hasil Verifikasi untuk spesifikasi Liveness Condition*

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

IV.2.2.2 Verifikasi Algoritma Dijkstra Semaphore

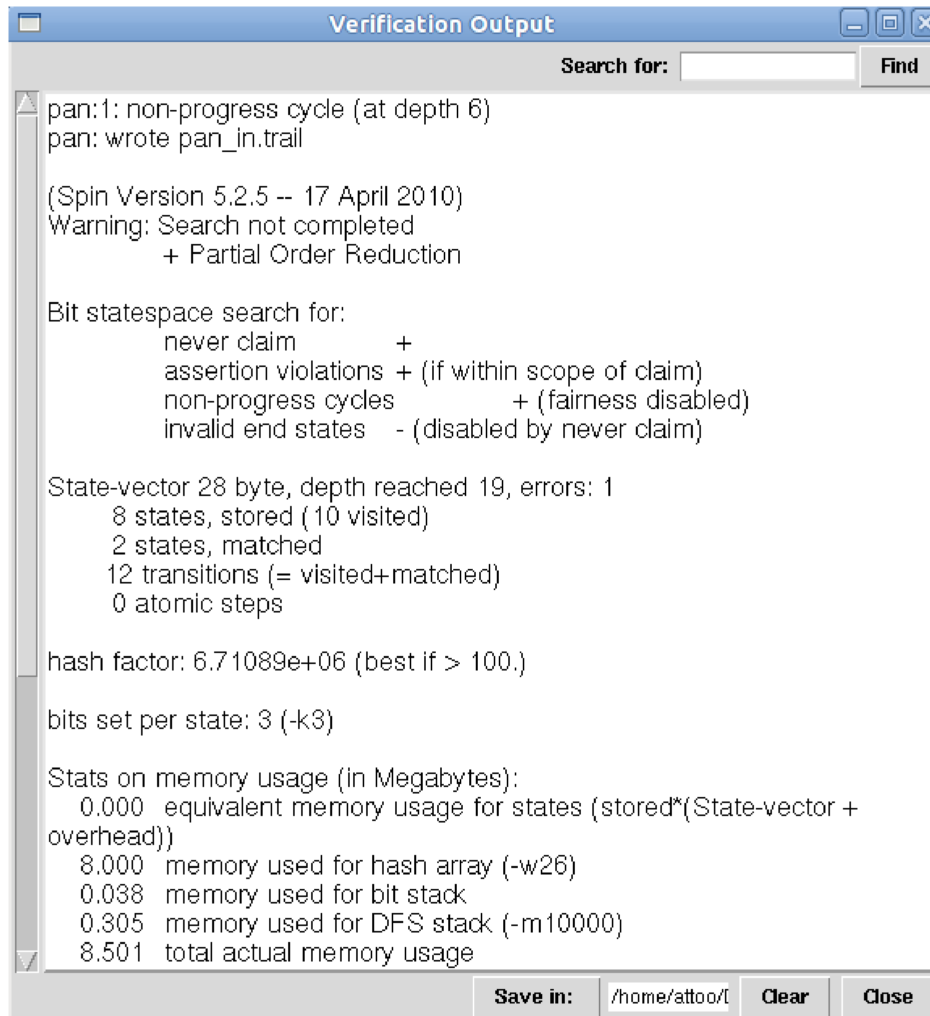
Verifikasi untuk spesifikasi pada kondisi *Safety (state properties)* dengan option *assertion* dan *invalid endstates* dihidupkan, modus pencariannya *supertrace/bitstate* , maka didapat hasil verifikasi sebagai berikut.



Gambar 4.17 *Hasil Verifikasi untuk spesifikasi Safety Condition*

Dari hasil verifikasi ini dapat dilihat bahwa algoritma Dijkstra Semaphore tidak bermasalah dalam *assertion violation* dan juga pada state yang dibangun, artinya tidak ada proses yang melanggar *mutual exclusion*.

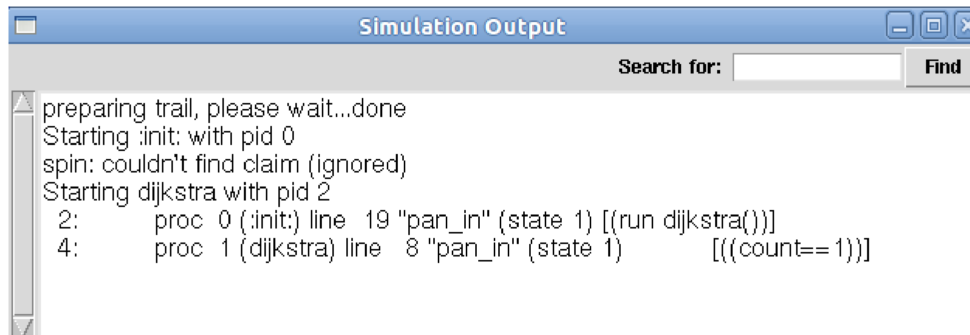
Verifikasi untuk spesifikasi pada kondisi *Liveness (cycles/seguence)* dengan option *non–progress cycles* dihidupkan, modus pencariannya *supertrace/bitstate,* maka didapat hasil verifikasi sebagai berikut.



Gambar 4.18 *Hasil Verifikasi untuk spesifikasi Liveness Condition*

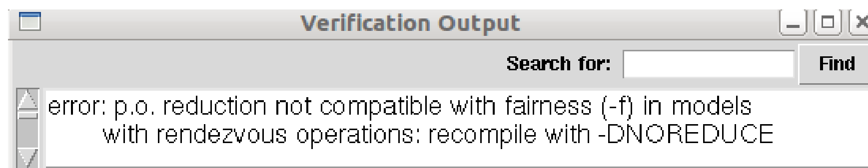
Dari hasil verifikasi ini dapat dilihat bahwa algoritma Dijkstra Semaphore ini mengalami `non-progress cycle (at depth 6)`, sehingga terjadi pelanggaran *mutual exclusion*. Dimana jika terdapat *non-progress cycles* , artinya prinsip *fairness* dalam memasuki *critical section* tidak ada. Untuk lebih jelasnya dapat dilihat dalam hasil simulasi dibawah ini.

Hasil simulasi baru setelah verifikasi :



Gambar 4.19 *Hasil akhir simulasi baru untuk Liveness Condition*

Verifikasi untuk spesifikasi pada kondisi *Liveness (cycles/seguence)* dengan dengan penambahan option *with weak fairness* dihidupkan, maka hasil verifikasi didapat sebagai berikut.



Gambar 4.20 *Hasil Verifikasi untuk spesifikasi Liveness Condition*

Dari hasil verifikasi ini dapat dilihat bahwa pada algoritma Dijkstra Semaphore ini juga tidak sesuai dengan prinsip *fairness* dalam model.

BAB V

PENUTUP

V.1. Kesimpulan

Setelah melakukan pembuktian algoritma yang menjamin sifat-sifat *mutual exclusion* tersebut serta simulasi dan verifikasi terhadap algoritma *distributed mutual exclusion*, maka dapat diambil beberapa kesimpulan sebagai berikut.

1. Algoritma Peterson menyelesaikan masalah *distributed mutual exclusion* dan menjamin sifat *safety mutex, liverness* dan *fairness.*
2. Algoritma Dijkstra Semaphore tidak dapat menjamin sifat *liverness* dan *fairness,* tetapi masih dapat digunakan untuk menyelesaikan masalah *distributed mutual exclusion* pada situasi tertentu secara hati-hati.
3. Algoritma Dijkstra Semaphore tidak mengalami pelanggaran *mutual exclusion* pada kondisi *Safety (state properties)* tetapi pada kondisi *Liveness (cycles/seguence)* terdapat pelanggaran *non-progress cycle.*
4. Algoritma Peterson tidak mengalami pelanggaran *mutual exclusion* pada kondisi *Safety (state properties)* dan *Liveness (cycles/seguence)*
5. Bila simulasi yang dilakukan berhasil, belum tentu bahwa algoritma tersebut setelah diverifikasi hasil kerjanya juga sudah baik.

V.2. Saran

Untuk penelitian lebih lanjut dapat dilakukan pengkajian mengenai PROMELA/SPIN untuk penyelesaian mutual exclusion dengan pengembangan masalah *k*-mutex, generalition mutex, group mutex, dan masalah mutual exclusion lainnya pada sistem terdistribusi. Demikian juga penggunaan PROMELA/SPIN untuk berbagai algoritma masalah-masalah komputasional.

DAFTAR PUSTAKA

Ardiansyah. *Mata Kuliah: Sistem Operasi*. http://blog.uad.ac.id/mas_woko/files/2009/11/so-pertemuan4.pdf, (diakses 12 Maret 2011).

Anderson, Sheila and friends. (1998). *A Tutorial in Model Checking*. CIS 841 Lecture website, University of Kansas. http://www.cis.ksu.edu, (diakses 25 Maret 2011).

Bayu U. Muhammad dan K. Yodhi. *Coordination and Agreement*. Jurusan Teknik Elektro FT UGM. Yogyakarta.

Dwi Priyanto, Nico Anandito, dan Sactio Swastioyono. *Bab 19. Solusi Critical Section*.http://bebas.vlsm.org/v06/Kuliah/SistemOperasi/2008/240/19._Solusi_Critical_Section-1.pdf, (diakses 12 Maret 2011).

Dwyer, Matt and Hatcliff, John. (2001). *Lecture SPIN-INTRO:Introduction To SPIN*. CIS842 Lecture presentation at University of Kansas. http://www.cis.ksu.edu, (diakses 25 Maret 2011).

Fariza, Arna. (2008). *Diktat SO: Bab 5. Sinkronisasi Proses*. TI- PENS ITS. http://lecturer.eepis-its.edu/~arna/Diktat_SO/5.Sinkronisasi%20Proses.pdf, (diakses 12 Maret 2011).

Lynch, Nancy A. (1996). *Distributed Algorithms*. Morgan Kaufman Publishers, Inc. San Franscisco, California.

Masyarakat Digital Gotong Royong (MDGR). (2004). *Pengantar Sistem Operasi Komputer, Plus Studi Kasus Kernel Linux*. http://kambing.ui.ac.id/bebas/v06/Kuliah/SistemOperasi/BUKU/SistemOperasi-2.0.pdf, (diakses 25 Maret 2011).

Nasrun, M. (2004). *Verifikasi Beberapa Algoritma Penanganan Proses untuk Penyelesaian Masalah Mutual Exclusion dengan Promela dan Spin*. Departemen Teknik Elektro, Institut Teknologi Bandung. Bandung.

Rahardjo, Budi. (2005). *Pengantar Metoda Formal*. Teknik Elektro Institut Teknologi Bandung. Bandung.

Velazquez, Martin G. (1993). *A Survey of Distributed Mutual Exclusion Algorithms*. Technical Report of Departemnt of Computer Science Colorado State University.