Analisis Penerapan *Semaphore* dalam Mengatasi Masalah Sinkronisasi *Dining Philosophers.*

Ramdani[1]**, D. Lesmiadi**[1]**, K. Santika**[1]**, M. Nasrun**[1]**, Irzaman**[2]
[1] Jurusan Informatika, FT, Jl. Dipati Ukur Bandung
[2] Jurusan Fisika, FMIPA IPB, Jl. Raya Pajajaran Bogor

ABSTRACT

Initially most people wear simple synchronization concepts which supported by hardware, like usage of routine usage or interrupt which implementation could have by hardware. In the year 1967, Djikstra raise a concept of an integer variable to count to the number of process which is active or which is not is active. Type of this variable is named by semaphore. semaphore is also used for synchronization in communications between device. of this journal semaphore used to finish the problem of dining philosopher syncronization. Dining philosopher itself is situation where 5 philosopher sit in dining table to eat spaghetti, each every philosopher given one spaghetti saucer and given one chopstick, to eat the the spaghetti required by two chopstick to finish this problem hence this semaphore variable is applied in each chopstick so that chopstick can kepilisopher share which its.

Keyword : *philosophers, Dining Philosophers, Semaphore.*

ABSTRAK

Pada awalnya kebanyakan orang memakai konsep-konsep sinkronisasi yang sederhana yang didukung oleh *hardware*, seperti pemakaian *interrupt* atau pemakaian rutin-rutin yang mungkin telah diimplementasi oleh *hardware*. Pada tahun 1967, Djikstra mengajukan suatu konsep pememakai suatu *variable* integer untuk menghitung banyaknya proses yang sedang aktif atau yang sedang tidak aktif. Tipe dari *variable* ini dinamakan *semaphore*. kebanyakkan *semaphore* juga digunakan untuk sinkronisasi dalam komunikasi antar *device* perangkat.Pada jurnal ini *semaphore* digunakan untuk menyelesaikan masalah singkronisasi *dining philosophers.Dining philosophers* tu sendiri adalah situasi dimana 5 *philosopher* duduk di meja makan untuk makan spageti, setiap *philosopher* diberi satu piring spageti dan diberi satu sumpit, untuk memakan spageti tersebut dibutuhkan dua sumpit untuk menyelesaikan masalah ini maka *variable semaphore* ini diterapkan pada setiap sumpit agar sumpit bisa di *share* ke-*philosopher* yang lainya.

Kata kunci : *philosophers, Dining Philosophers, Semaphore.*

1. Pendahuluan

Sinkronisasi diperlukan untuk menghindari terjadinya ketidak konsistenan data akibat adanya akses data secara konkuren. Akses-akses yang dilakukan secara bersama-sama ke data yang sama, dapat menyebabkan data menjadi tidak konsisten. Konsisten artinya data yang sebenarnya tidak berubah walaupun melalui beberapa proses. Pada tahun 1967, Djikstra mengajukan suatu konsep variabel integer untuk menghitung banyaknya proses yang sedang aktif atau yang sedang tidak aktif. Tipe dari *variable* ini dinamakan *semaphore*.[1]. Dibandingkan dengan *variable* lain *variable semaphore* memiliki kelebihan dimana *variable* bertipe integer ini diakses dengan 2 standar operasi *atomic,* yaitu *wait* dan *signal* yang tidak dimiliki oleh *variable* integer yang lainya.

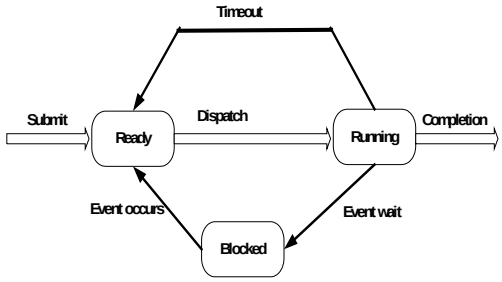
Semaphore juga digunakan untuk sinkronisasi dalam komunikasi antar *device* (perangkat keras), nilai awal dari *semaphore* tersebut menunjukkan berapa banyak proses yang boleh memasuki *critical section* dari suatu program. Biasanya untuk mendukung sifat *mutual exclusive*, nilai ini diberi

2. Pembahasan

Sistem operasi adalah antarmuka utama yang digunakan dalam berhubungan dengan sistem komputer[1]. Sistem operasi berfungsi untuk mengatur dan mengawasi penggunaan perangkat keras oleh berbagai program aplikasi serta para pengguna. Untuk menghindari konflik yang terjadi pada saat pengguna menggunakan sumber daya yang sama, sistem operasi mengatur pengguna mana yang dapat mengakses suatu sumber daya. Sistem operasi juga sering disebut *resource allocator* [2].

Proses adalah suatu program yang sedang dalam keadaan eksekusi[3].

Proses melewati serangkaian *state*. Beragam kejadian dapat menyebabkan perubaha *state* proses. Proses dapat berada di salah satu dari tiga *state* dasar [1].



Gambar 1. Diagram Status Proses

Hubungan ketiga *state* dasar dapat dilihat pada gambar 1. Proses yang baru dibuat akan mempunyai *state ready*. Proses ber-*state running* menjadi *blocked* karena sumber daya yang diminta belum tersedia sehingga menunggu kejadian muncul. Proses menunggu kejadian alokasi sumber daya (*event wait*). Proses ber-*state running* menjadi *ready* karena penjadwalan memutuskan aksekusi proses lain karena jatah waktu untuk proses tersebut telah habis (*time out*). Proses ber-*state blocked* menjadi *ready* saat sumber daya yang diperlukan telah tersedia (*event occurs*). Proses ber-*state ready* menjadi *running* karena penjadwalan memutuskan penggunaan proses[1].

Kongkurensi merupakan landasan umum perancangan sistem operasi. Proses-proses disebut kongkurensi jika proses-proses lebih dari satu proses berada pada waktu yang sama [1]. Prinsip-prinsip Kongkurensi, kongkurensi meliputi hal-hal berikut [2] :

1. Alokasi waktu pemroses untuk proses-proses.
2. Pemakaian bersama dan persaingan untuk mendapatkan sumber daya.
3. Komunikasi antar proses.
4. Sinkronisasi aktivitas banyak proses.

Race Condition adalah situasi di mana beberapa proses mengakses dan memanipulasi data bersama pada saat besamaan. Untuk mencegah *race condition*, proses-proses yang berjalan besamaan harus disinkronisasi [4].

Critical Section adalah sebuah segmen kode proses yang mana sumber daya bersama diakses.

Secara umum, penyelesaian *critical section* harus memenuhi 3 syarat [5] :

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 megerjakan *critical sectio*n, dan ada beberapa proses yag akan masuk ke *critical sectio*n, maka hanya proses-proses yang sedang berada pada *entry-section* saja yang boleh berkompetisi untuk mengerjakan *critical section*.
3. *Bounded waiting*. Besarnya waktu tunggu dari suatu proses yang akan memasuki *critical section* sejak proses itu meminta ijin untuk mengerjakan *critical sectio*n, hingga permintaan itu dipenuhi

Mutual exclusion adalah jaminan hanya satu proses yang mengakses sumber daya pada satu interval waktu tertentu.

Thread, atau kadang-kadang disebut proses ringan (*lightweight*), adalah unit dasar dari utilisasi CPU. Di dalamnya terdapat ID *thread*, program *counter*, *register*, dan *stack*. Dan saling berbagi dengan *thread* lain dalam proses yang sama [6].

Managemen *Thread Java* menyediakan beberapa fasilitas untuk mengatur *thread* - *thread*, diantaranya adalah:

1. *Suspend*( ): berfungsi untuk menunda eksekusi dari *thread* yang sedang berjalan.
2. *Sleep*( ): berfungsi untuk menempatkan *thread* yang sedang berjalan untuk tidur dalam beberapa waktu.
3. *Resume*( ): hasil eksekusi dari *thread* yang sedang ditunda.
4. *Stop*( ): menghentikan eksekusi dari sebuah *thread*; sekali *thread* telah dihentikan dia tidak akan memulainya lagi.

Semaphore adalah pendekatan yang diajukan oleh Djikstra, dengan prinsip bahwa dua proses atau lebih dapat bekerja sama dengan menggunakan penanda-penanda sederhana. Seperti proses dapat dipaksa berhenti pada suatu saat, sampai proses mendapatkan penanda tertentu. Variabel khusus untuk penanda ini disebut *semaphore*. *Semaphore* mempunyai sifat, yaitu [3,4]:

1. Terdapat dua operasi terhadap *semaphore*, yaitu Down dan Up. Usulan asli yang disampaikan Djikstra adalah operasi P (*wait*) dan V (*signal*).

Operasi-operasi yang diwakili dengan P (*wait*) dan V(*signal*) dapat dilihat pada persamaan (1), sebagai berikut :

Wait(S) : while S ≤0 do no_op;
S:=S – 1;

Signal : S:=S + 1;

(1)

Implementasi, Algoritma *semaphore* Djikstra dapat dilihat pada Gambar 2.

```
type semaphore = record
        Value: integer;
        L: list of process;
        End;
Var S: semaphore;
Operasi-operasi pada semaphore
Wait (S): S.value := S.value-1;
If  S.value < 0 then
Begin
Tambahkan proses ini ke S.L.block;
End;
Signal (S) :        S.value := S.value+1;
If S.value ≤ 0 then
Begin
Hapus suatu proses dari S.L.wekup (P);
End;
```

Gambar 2. Algoritma *semaphore* Djikstra

- **Subrutin *Wait***

Seperti yang telah dijelaskan di atas, bahwa di dalam *subrutin* ini, proses akan memeriksa harga dari *semaphor*e, apabila harganya 0 atau kurang maka proses akan menunggu, sebaliknya jika lebih dari 0, maka proses akan mengurangi nilai dari *semaphore* tersebut dan menjalankan operasi yang lain. Bila diperhatikan lebih kritis lagi, pernyataan "menunggu" sebenarnya masih abstrak. Bagaimanakah cara proses tersebut menunggu. Cara proses menunggu dapat dibagi menjadi dua :

1. *spinlock waiting*
2. *non-spinlock waiting*

Spinlock waiting berarti proses tersebut menunggu dengan cara menjalankan perintah-perintah yang tidak ada artinya. Dengan kata lain proses masih *running state* di dalam *spinlock waiting*.. Berbeda dengan *spinlock waiting*, *non-spinlock waiting*, memanfaatkan fasilitas sistem operasi. Proses yang melakukan *non-spinlock waiting* akan mem-*block* dirinya sendiri dan secara otomatis akan membawa proses tersebut ke dalam *waiting queue*. Di dalam *waiting queue* ini proses tidak aktif dan menunggu sampai ada proses lain yang membangunkan dia sehingga membawanya ke *ready queue*.

Subrutin *Signal*

Karena subrutin *wait* memiliki 2 versi maka hal ini juga berpengaruh kepada subrutin *signal*. Subrutin *signal* akan terdiri dari 2 versi sesuai dengan yang ada di subrutin *wait*. Yang membedakan antara *signal spinlock* dengan *Non-Spinlock* adalah fungsi *Notify()* yang digunakan oleh subrutin *signal Non-Spinlock* yang lebih efektif digunakan dalam membangunkan atau memeberi *signal* suatua proses.

Ada 2 macam *semaphore* yang cukup umum, yaitu :

1. *Binary semaphore*
2. *Counting semaphore*

Binary semaphore adalah *semaphore* yang bernilai hanya 1 dan 0. Sedangkan *Counting semaphore* adalah *semaphore* yang dapat bernilai 1 dan 0 dan nilai integer yang lainnya.

- ***Dining Philosopher***

Pada tahun 1965, Djikstra menyelesaikan sebuah masalah sinkronisasi yang beliau sebut dengan *dining philisophers problem*. *Dining philosopherss* dapat diuraikan sebagai berikut: Lima orang *philosophers* duduk mengelilingi sebuah meja bundar. Masing-masing *philosophers* mempunyai sepiring spageti. Spageti-spageti tersebut sangat licin dan membutuhkan dua sumpit untuk memakannya. Diantara sepiring spageti terdapat satu sumpit, dapat dilihat pada gambar3.



Gambar3. Situasi *Dining philosophers*

Kehidupan para *philosophers* terdiri dari dua periode, yaitu makan atau berpikir. Ketika seorang *philosophers* lapar, *philosophers* berusaha untuk mendapatkan sumpit kiri dan sumpit kanan sekaligus. Jika sukses dalam mengambil kedua sumpit, *philosophers* tersebut makan untuk sementara waktu, kemudian meletakkan kedua sumpit dan melanjutkan berpikir. Pertanyaan kuncinya adalah, dapatkah masing-masing *philosophers* tersebut melakukan prosesnya dengan benar dan tidak pernah mengalami kebuntuan.

ada dua masalah yang di hadapi *dining philosopher* yaitu :

1. *Deadlock*
2. *Starvation*

Solusi *Dining philosophers*

Proses di atas bisa diatasi "jika *philosophers* dapat saja menunggu sebuah waktu acak sebagai pengganti waktu yang sama setelah tidak dapat mengambil sumpit kiri dan kanan, kesempatan bahwa segala sesuatau akan berlanjut dalam kemandegan untuk beberapa waktu adalah sangat *kecil*." Pemikiran seperti itu adalah benar, tapi beberapa aplikasi mengirimkan sebuah solusi yang selalu bekerja dan tidak ada kesalahan.

Sebelum mulai mengambil sumpit, seorang *philosophers* melakukan *DOWN* di *mutex*. Setelah mengambil sumpit *philosophers* harus melakukan *UP* di *mutex*. Dari segi teori, solusi ini cukup memadai. Dari segi praktek, solusi ini tetap memiliki masalah. Hanya ada satu *philosophers* yang dapat makan spageti dalam berbagai kesempatan. Dengan lima buah sumpit, seharusnya bisa menyaksikan dua orang *philosophers* makan spageti pada saat bersamaan.

Solusi yang diberikan di atas benar dan juga mengizinkan jumlah maksimum kegiatan paralel untuk sejumlah *philosophers* yang berubah-ubah ini menggunakan sebuah *array state*, untuk merekam status seorang *philosophers* apakah sedang makan (*eating*), berpikir (*think*), atau sedang lapar (*hungry*) karena sedang berusaha mengambil sumpit. Seorang *philosophers* hanya dapat berstatus makan (*eating*) jika tidak ada tetangganya yang sedang makan. Tetangga seorang *philosophers* didefinisikan oleh *LEFT* dan *RIGHT.*

Dengan kata lain, jika i = 2, maka tetangga kirinya (*LEFT*) = 1 dan tetangga kanannya (*RIGHT*) = 3. Program ini menggunakan *sebuah* array dari *semaphore* yang lapar (*hungry*) dapat ditahan jika sumpit kiri atau kanannya sedang dipakai tetangganya. Catatan bahwa masing-masing proses menjalankan prosedur *philosophers* sebagai kode utama, tetapi prosedur yang lain seperti ambil_sumpit, dan *test* adalah prosedur biasa pada proses-proses yang terpisah.

Pengujian Program

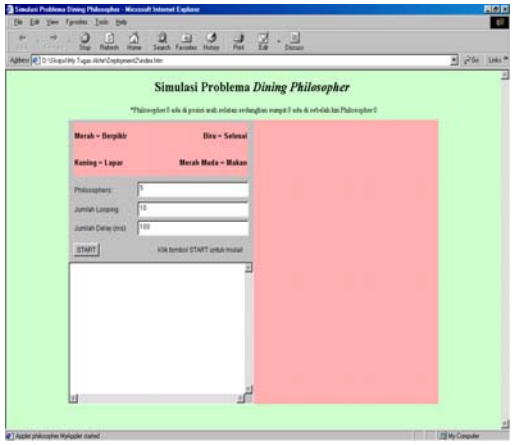
Philosopher diberi nama dari 0 sampai jumlah *philosopher* dikurangi 1 dan posisi *philosopher* 0 adalah pada posisi selatan. *Philosopher* 1 dan seterusnya terletak berlawanan dengan arah jarum jam. Sumpit diberi nama A, B … Huruf ke jumlah *philosopher*

dengan sumpit A ada di sebelah kiri *philosopher* 0, sumpit B dan seterusnya terletak berlawanan dengan arah jarum jam. Untuk mengamati simulasi, tentukan *delay* yang cukup sehingga perpindahan status dapat diamati dengan jelas (*default*-nya 100 ms), untuk proses yang lebih lambat *delay* dapat diisi dengan nilai yang lebih besar sedangkan untuk proses yang lebih cepat, *delay* dapat diisi dengan nilai yang lebih kecil.

Perhatikan status *philosopher* warna kuning (lapar). *Philosopher* akan berganti status menjadi warna *pink* (makan) jika sumpit sebelah kiri dan sebelah kanan tersedia dan pada gambar akan terlihat satu persatu sumpit di angkat (dalam gambar tampak menghilang) berurutan sebelah kanan dahulu baru sebelah kiri.

Philosopher akan berganti status menjadi warna biru (selesai) jika kedua sumpit telah diletakkan berurutan, yaitu sebelah kanan dahulu baru sebelah kiri. Segera setelah sebuah sumpit diletakkan, *philosopher* disebelahnya yang statusnya kuning (lapar) dan sumpit sebelah lainnya tersedia akan berubah status menjadi *pink* (makan) kecuali *philosopher* di sebelah lainnya statusnya kuning juga. Jika keadaan ini terjadi, maka hanya salah satu *philosopher* yang akan berubah statusnya, *philosopher* yang lain akan kembali menunggu sampai kedua sumpit di sebelah kiri dan kanannya tersedia.

Tampilan program dapat dilihat pada gambar 4



Gambar 4. Tampilan Awal Program

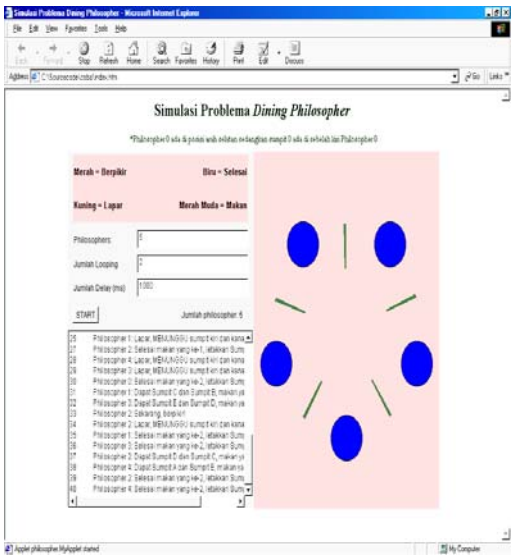
Simulasi dimulai dengan 5 *philosopher*

Tabel 1. Proses Makan *philosophers*

Philosophers	**Makan ke pada iterasi**	
	1	**2**
Philosophers 0	33	39
Philosophers 1	11	27
Philosophers 2	16	31
Philosophers 3	12	28
Philosophers 4	15	32

Proses berjalan seperti yang diharapakan tidak terjadi *Deadlock* dan *starvation* proses tidak mengalami kemandegan (berhenti sebelum waktu proses selesai) dan proses semuanya dilayani. Walaupun proses berjalan dengan benar, ada proses proses yang mengarah ke salah satu kondisi kritis yaitu *starvation.* Dapat di lihat dari table hasil pengujian Tabel 1, *Philosophers* yang pertama dilayani adalah *philosophers* 3 pada iterasi 12,dan 12 sedangakan philosopher 0, dilayani pada proses terakhir yaiotu iterasi 33 dan 39.

Tampilan selesai dapat dilihat pada gambar 5.



Gambar 5. Tampilan Akhir Program

Pertama-tama semua *philosophers* berfikir setelah berfikir semua *philosophers* merasakan lapar dan menunggu sumpit kiri dan sumpit kanan tersedia di *waiting thread* setiap *philosopher* diwakili oleh satu *thread* disini terjadi *rece condition* karena pemanggilan fungsi *notify*( ) secara bersamaan diwaktu yang sama pula, itu terjadi ketika *waiting thread* mendapat *interrupt*dari beberapa *thread* yang lain. Setelah *philosophers* mendapat kedua sumpit dia bergerak dari *waiting thread* ke *waiting set* yang merupan antrian pada program ini (*entry section*) menuju ke *ready queue* dan kemudian melakukan proses makan,. Permasalahan waktu yang sama itu terjadi akibat beberapa *thread* ingin mendapatkan fungsi V( ) atau *signal* untuk memanggil fungsi *notify*( ). Setelah makan selesai fungsi *notify*( ) sudah mengerjakan prosesnya dan *waiting set* kosong maka tidak ada proses yang terjadi sampai *thread* yang lain masuk kembali ke *waiting set*. kemudian *thread* yang sudah melakukan proses makan masuk ke fungsi P( ) atau melakukan *wait* di *catch block* dan menunggu kembali sampai mendapatkan *signal*. Kehilangan *signal* pada *mutual exclusion* akan membawa ke kondisi *deadlock* dan *starvation*.

Kesimpulan

Algoritma yang dibangun ternyata efektif karena memberikan solusi setelah di implementasikan ke bentuk program di mana kondisi *deadlock* dan *starvation* tidak terjadi. Algoritma tersebut tidak melakukan pelanggaran pada *critical section*, *Dining philosophers* merupakan masalah yang timbul disebabkan kurangnya sumber daya yang dimiliki setiap proses. Untuk mengatasi masalah *critical section* yang di hadapi *dining philosophers* dapat digunakan berbagai solusi *software*. Namun masalah yang akan timbul dengan solusi *software* biasa adalah solusi *software* tidak mampu menangani masalah yang lebih berat dari *critical section*. Sedangkan solusi *software* dengan menggunakan *Semaphores* mampu menanganinya, terbukti dari *software* yang penulis buat dengan menggunakan *semaphore* dapat mengatasi masalah sinkronisasi *dining philosophers*, proses *dining philosophers* berjalan dengan benar juga dapat menghidari kondisi *deadlock* dan *starvation* yang menjadi masalah utama pada sinkronisari *dining philosophers.*

Daftar Pustaka

[1] B. B. Hariyanto, *Sistem Operasi*, Bandung, hal. 19 - 32 (1999).

[2] B. Wibowo, A. Setiawan, Baya, A. Budi, Heriyanto, dan M. Rusdi, *Sistem Operasi Bahan Kuliah IKI-20230 BAB I*, hal. 1-3 (2003).

[3] S. Kusumadewi, "*Sistem operasi*", J&J Learning Yogyakarta,hal 51-53 (2000).

[4] A. W. Jatmiko, A. Pratomo, D. Kurniawan, S. Adisasmito, Z. Agni., *Sistem Operasi Bahan Kuliah IKI-20230 BAB III*, hal. 73-74 (2003).

[5] M. Nasrun, *Verifikasi beberapa algoritma penanganan proses untuk penyelesaian masalah mutual exclusion*, hal 1 (2003).

[6] I. Agung, A. *Khumaidi*, Arifulah, A.S. Baihaki, K.F. Christian, Daeli, E. Nugroho, E. Seno, Habrar, H. Sahlan, *Sistem Operasi Bahan Kuliah IKI-20230 BAB II*, hal. 38-61 (2003).