DEADLOCK PADA DISTRIBUSI DATA DAN PEMECAHANNYA

Dion Jogi Parlinggoman 13509045
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13509045@std.stei.itb.ac.id

***Abstract*— Bahasan graf merupakan bahasan yang dapat menjadi pusat perhatian, karena mampu memodelkan suatu sistem atau cara kerja dari sekumpulan jaringan yang mempunyai perilaku untuk mencapai tujuannya masing – masing, seperti perilaku dalam jaringan komunikasi, transportasi, ilmu komputer, dan lain sebagainya. Salah satu aplikasi graf pada sistem operasi pendistribusian data adalah graf yang digunakan untuk melakukan pendeteksian dan pencegahan *deadlock*. *Deadlock* adalah suatu kondisi dimana sistem tidak berjalan lagi karena setiap proses yang ada menunggu suatu kejadian yang hanya dapat dilakukan oleh proses lain dalam himpunan tersebut.**

Sistem operasi ialah pengelolaan terhadap sumber daya untuk mencapai tujuan dengan mengelola permintaan terhadap sumber daya secara efisien dan juga mengatur eksekusi aplikasi / keluaran. Fungsi ini dikenal juga sebagai program pengendali (*control program*). Sistem operasi merupakan suatu bagian proses yang berjalan setiap saat dan komponen - komponen dasar sistem operasi adalah implementasi graf.

Dalam hal ini akan dibahas mengenai implementasi graf dalam sistem operasi, yaitu penggunaannya untuk penanganan *deadlock* pada distribusi data. Graf alokasi sumber daya adalah bentuk dalam mendeteksi masalah *deadlock* pada sistem operasi pendistribusian data. Mekanisme hubungan dari proses - proses dan sumber daya yang dibutuhkan atau digunakan itu dapat di diwakilkan dan digambarkan dengan graf alokasi sumber daya.

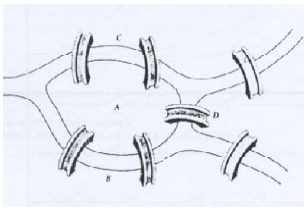
***Kata kunci*: sistem operasi, *deadlock*, sumber daya, distribusi data.**

1. PENDAHULUAN

1.1. Graf

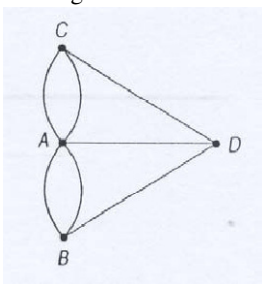
1.1.1. Sejarah Graf

Graf dipakai pertama kali oleh seorang matematikawan Swis pada tahun 1763 untuk memecahkan teka-teki jembatan Konigsberg. Di kota Konigsberg –Jerman Timur– terdapat sungai Pregal yang dibelah dua oleh Pulau Kneipof. Daratan yang dipisahkan oleh sungai tersebut dihubungkan oleh tujuh buah jembatan. Teka-tekinya adalah **"apakah mungkin melalui ketujuh jembatan tersebut dan kembali ke tempat semula dengan masing-masing jembatan dilalui tepat satu kali ?"**



Gambar 1-1 Pemodelan jembatan pada kota Konigsberg

Sebelum Euler memodelkan masalah ini ke dalam graf dan mengemukakan solusinya, kebanyakan orang sepakat bahwa tidak mungkin kembali ke tempat semula namun mereka tidak mampu menjelaskan mengapa. Euler memodelkan daratan dengan titik yang disebut sebagai simpul dan jembatan yang menghubungkannya sebagai garis yang disebut sebagai sisi.



Gambar 1-2 Graf yang merepresentasikan jembatan Konigsberg

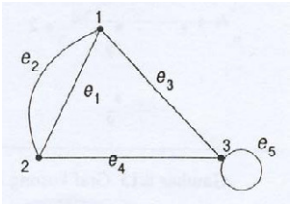
Jawaban Euler adalah **"orang tidak mungkin melalui ketujuh jembatan tersebut masing-masing satu kali dan kembali lagi ke tempat semula jika jumlah sisi dari masing-masing simpul tidak seluruhnya genap."** Jika masing-masing simpul memiliki jumlah sisi genap maka dengan melalui masing - masing sisi satu kali kita dapat kembali ke tempat semula.

Dari gambar di atas tampak bahwa simpul-simpul dari graf pemodelan jembatan Konigsberg memiliki sisi berjumlah ganjil, jadi orang tidak mungkin kembali ke tempat semula.

1.1.2. Definisi Graf

Graf didefinisikan sebagai pasangan himpunan (V,E), ditulis dengan notasi G = (V,E), yang dalam hal ini V adalah himpunan tidak kosong dari simpul-simpul (*vertices* atau *node*) = $\{v_1, v_2, \ldots v_n\}$ dan E adalah himpunan sisi-sisi (*edges* atau *arcs*) yang menguhubungkan sepasang simpul = $\{e_1, e_2, \ldots e_n\}$.

Simpul pada graf dinomori dengan huruf atau bilangan asli atau gabungan keduanya. Sedangkan sisi yang menghubungkan simpul u dengan simpul v dinyatakan dengan pasangan (u,v) atau dengan lambang e_i dimana i = himpunan bilangan asli = {1,2,3,...} .



Gambar 1-3 Penamaan Graf

Secara geometri graf direpresentasikan sebagai himpunan titik yang dihubungkan dengan himpunan garis dalam ruang dua dimensi.

1.1.3. Jenis – jenis Graf

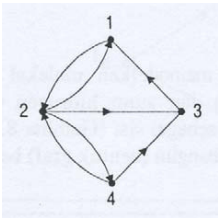
Berdasarkan jenis sisinya graf digolongkan menjadi dua jenis :

1. Graf sederhana yaitu graf yang tidak memiliki sisi ganda maupun sisi loop.
2. Graf tidak sederhana yaitu graf yang memiliki sisi ganda atau sisi loop. Graf tidak sederhana dibedakan menjadi dua yaitu graf ganda dan graf semu. Graf ganda yaitu graf yang memiliki sisi ganda. Sisi ganda adalah sekumpulan sisi yang menghubungkan sepasang simpul yang sama. Dengan kata lain, bila sepasang simpul dihubungkan oleh lebih dari satu sisi, maka sisi-sisi itu disebut sisi ganda. Graf semu yaitu graf yang memiliki sisi loop. Loop adalah sisi yang menghubungkan sebuah simpul dengan dirinya sendiri.

Jumlah simpul pada graf disebut sebagai kardinalitas graf, dan dinyatakan dengan $n = | V |$, dan jumlah sisi dinyatakan dengan $m = | E |$.

Ada dua jenis graf berdasarkan orientasi arah dari sisi-sisi graf:

1. Graf tak berarah, yaitu graf yang sisi-sisinya tidak memiliki arah.
2. Graf Berarah, yaitu graf yang sisi-sisinya memiliki orientasi arah.



Gambar 1-4 Graf Berarah

Pada graf jenis ini, sisi yang menghubungkan simpul 2 ke simpul 1 tidak sama dengan sisi yang menghubungkan simpul 1 ke simpul 2 karena orientasi arahnya berbeda.

1.2. Distribusi Data

Database terdistribusi adalah database yang disimpan pada beberapa komputer didistribusi dalam sebuah sistem terdistribusi melalui media komunikasi seperti high speed buses atau telepone line. Sistem database terdistribusi adalah berisi kumpulan site dan mengeksekusikan transaksi lokal (mengakses data pada satu site) & transaksi global (mengakses data pada site berbeda).

Contoh :

Transaksi lokal : transaksi menambahkan dana pada nomor rekening 1112234 yang berada di cabang margonda. Transaksi ditentukan pada cabang margonda.

Transaksi global : transaksi transfer dari rekening 1112234 ke rekening 2223410 yang berada di kramat jati (rekening didua site berbeda telah diakses sebagai hasil dari eksekusinya)

Site-site dalam database terdistribusi dihubungkan secara fisik dengan berbagai cara. Beberapa topologi digambarkan sebagai sebuah graf. Beberapa bentuk :

1. Fully connected network
   Apabila salah satu *node* rusak, bagian yang lain masih dapat berjalan (biaya mahal). Kontrol manajemen tidak terjamin.
2. Partially connected network
   Reliability rendah dan biaya dapat ditekan tetapi kontrol manajemen tidak terjamin.
3. Tree structured network
   Bersifat sentral dan kontrol manajemen lebih terjamin. Kalau *node* pusat rusak, semua akan rusak (setiap proses dimulai dari bawah).
4. Ring network (LAN)
   Apabila yang satu rusak, yang lain masih bisa berjalan. Kontrol manajemen kurang terjamin karena bersifat desentralisasi.
5. Star network (LAN)
   Kontrol manajemen lebih terjamin, karena bersifat sentral. Jika pusat mengalami kerusakan yang lain menjadi rusak.

Contoh Sistem terdistribusi:

- Internet = Interconnection Network
- Intranet Coorporation
- Mobile Computing
- Automated banking systems
- Tracking roaming cellular phones
- Global positioning systems
- Retail point-of-sale terminals
- Air-traffic control

Keuntungan dalam penggunaan sistem terdistribusi:

- *Resources sharing*: sumber daya dapat digunakan secara bersama / bergantian
- Distribusi fungsi: komputer memiliki kemampuan fungsi yang berbeda-beda
  - client/server
  - Host/terminal
  - Data gathering / data processing
- Distribusi beban/keseimbangan: pemberian tugas ke prosesor secukupnya sehingga unjuk kerja seluruh sistem teroptimasi.
- Replikasi kekuatan pemrosesan: independen *processors* bekerja untuk pekerjaan yang sama
  - Sistem terdistribusi terdiri dari kumpulan mikrokomputer yang memiliki kekuatan pemrosesan yang tidak dapat dicapai oleh supercomputer.

- *Reliability*: dalam sistem terdistribusi, apabila sebuah situs mengalami kegagalan, maka situs yang tersisa dapat melanjutkan operasi yang sedang berjalan. Hal ini menyebabkan reliabilitas sistem menjadi lebih baik.
- Pemisahan fisik : sistem yang menggantungkan pada fakta bahwa komputer secara fisik terpisah.
- Ekonomis : kumpulan *mikroprosesor* menawarkan harga/unjuk kerja yang lebih baik dari pada *mainframe*.
- Fleksibilitas : komputer yang berbeda dengan kemampuan yang berbeda dapat di share antar-*user*.

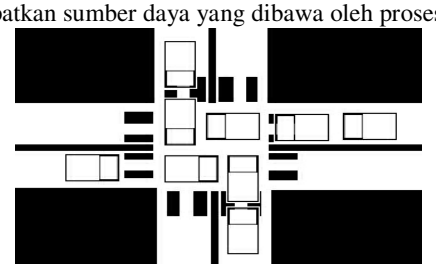
Tantangan yang dihadapi oleh sistem terdistribusi adalah

- *Heterogenity*
- *Scalability*: sistem tetap efektif meskipun terdapat peningkatan resource dan pengguna secara signifikan.
- *Openness*: memastikan sistem dapat diperluas dan mudah dalam pemeliharaan.
- *Security*:
  - *Confidentiality* (pencegahan terhadap hak akses oleh orang yang tidak berhak)
  - *Integrity* (pencegahan terhadap perubahan data)
  - *Availability* (pencegahan terhadap masalah ketersediaan)
- Konkurensi

1.3. *Deadlock*

1.3.1. Definisi *Deadlock*

Permasalahan deadlock terjadi karena sekumpulan proses - proses yang di - blok dimana setiap proses membawa sebuah sumber daya dan menunggu mendapatkan sumber daya yang dibawa oleh proses lain.



Gambar 1-5 Pemodelan *Deadlock*

Contohnya adalah terdapat satu jalur pada jalan. Mobil digambarkan sebagai proses yang sedang menuju sumber daya. Untuk mengatasi hal tersebut, beberapa mobil harus mundur. Sangat memungkinkan untuk terjadinya *starvation* yaitu kondisi proses tak akan mendapatkan sumber daya.

1.3.2. Model Sistem

Pada sistem terdapat beberapa sumber daya (*resource*) yang digunakan dalam proses - proses untuk menyelesaikan permasalahan dan mencapai tujuan. Setiap proses yang menggunakan sumber daya menjalankan urutan operasi sebagai berikut :

- meminta (*request*) : meminta sumber daya
- memakai (use) : memakai sumber daya
- melepaskan (*release*) : melepaskan sumber daya

1.3.3. Penyebab *Deadlock*

Deadlock terjadi bila terdapat empat kondisi berikut ini secara simultan.

a. *Mutual Exclusion* : hanya satu proses pada satu waktu yang dapat menggunakan sumber daya.

b. Genggam dan Tunggu (*Hold and Wait*) : suatu proses membawa sedikitnya satu sumber daya dan menunggu mendapatkan tambahan sumber daya baru yang dibawa oleh proses berikutnya.

c. *Non - Preemption* : sebuah sumber daya dapat dibebaskan dengan sukarela oleh proses yang memegangnya, setelah proses tersebut menyelesaikan bagiannya.

d. Menunggu Secara Sirkuler (*Circular Wait*) : Terdapat sekumpulan proses {*P*0, *P*1, …, *P*0} yang menunggu sumber daya dimana *P*0 menunggu sumber daya yang dibawa *P*1, *P*1 menunggu sumber daya yang dibawa *P*2, dan seterusnya, *Pn*–1 menunggu sumber daya yang dibawa oleh *P*n, dan *P*n menunggu sumber daya yang dibawa *P*0.

Ketiga syarat pertama merupakan syarat perlu (*necessary conditions*) bagi terjadinya *deadlock*. Keberadaan *deadlock* selalu mengindikasikan terpenuhinya kondisi - kondisi diatas. Tidak mungkin terjadi *deadlock* bila tidak ada ketiga kondisi tersebut. *Deadlock* terjadi berarti terdapat ketiga kondisi itu, tetapi adanya ketiga kondisi itu belum berarti terjadi *deadlock*.

Deadlock baru benar-benar terjadi bila syarat keempat terpenuhi. Kondisi keempat merupakan keharusan bagi terjadinya peristiwa *deadlock*. Bila salah satu saja dari kondisi tidak terpenuhi maka *deadlock* tidak terjadi.

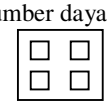
1.3.4. Graf Alokasi Sumber Daya

Deadlock dapat digambarkan dengan menggunakan graf berarah yang disebut *resource allocation graph*. Graf terdiri dari himpunan titik *V* dan garis *E*. Himpunan titik (vertex) *V* dibagi menjadi dua tipe yaitu himpunan proses yang aktif pada sistem $P = \{P1, P2, ..., Pn\}$ dan tipe sumber daya pada sistem $R = \{R1, R2, ..., Rm\}$. Garis berarah dari proses *P*i ke tipe sumber daya *R*j dinotasikan dengan *P*i → *R*j artinya proses *P*i meminta satu anggota dari tipe sumber daya *R*j dan sedang menunggu sumber daya tersebut. Garis berarah dari tipe sumber daya *R*j ke proses *P*i dinotasikan dengan *R*j → *P*i artinya satu anggota tipe sumber daya *R*j dialokasikan ke proses *P*i. Garis berarah *P*i → *R*j disebut *request edge* dan garis berarah *R*j → *P*i disebut *assignment edge*.

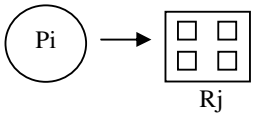
Notasi-notasi yang digunakan pada *resource allocation graph* adalah :

- Proses

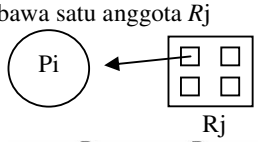
- Tipe sumber daya dengan 4 anggota

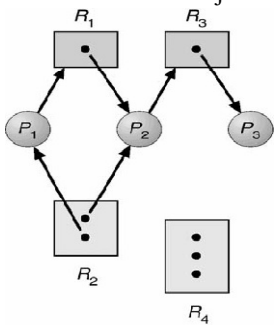


- *P*i meminta anggota dari *R*j



- *P*i membawa satu anggota *R*j





Gambar 1-6 Graf Alokasi Sumber Daya

Himpunan *P*, *R* dan *E* :

- *P* = {*P*1, *P*2, *P*3}
- *R* = {*R*1, *R*2, *R*3, *R*4}
- *E* = {*P*1 → *R*1, *P*2 → *R*3, *R*1 → *P*2, *R*2 → *P*2, *R*2 → *P*1, *R*3 → *P*3}

Anggota sumber daya :

- Satu anggota dari tipe sumber daya *R*1.
- Dua anggota dari tipe sumber daya *R*2.
- Satu anggota dari tipe sumber daya *R*3.
- Tiga anggota dari tipe sumber daya *R*4.

Status proses :

- Proses *P*1 membawa satu anggota tipe sumber daya *R*2 dan menunggu satu anggota tipe sumber daya *R*1.
- Proses *P*2 membawa satu anggota *R*1 dan *R*2 dan menunggu satu anggota tipe sumber daya *R*3.
- Proses *P*3 membawa satu anggota *R*3.

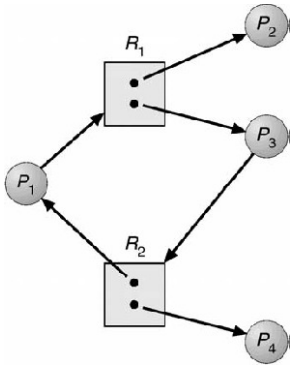
Fakta dasar dari *resource allocation graph* menunjukkan bahwa :

- Apabila pada graf tidak terdapat siklus maka tidak ada proses dalam sistem yang terjadi *deadlock*
- Apabila pada graf terdapat siklus sistem kemungkinan *deadlock* dengan ketentuan:
  - Jika pada setiap tipe sumber daya hanya terdapat satu anggota maka terjadi *deadlock*
  - Jika pada setiap tipe sumber daya terdapat beberapa anggota maka kemungkinan terjadi *deadlock*

Pada graf alokasi sumber daya Gambar 1-7 terdapat siklus :

$$P1 \rightarrow R1 \rightarrow P3 \rightarrow R3 \rightarrow P1$$

Akan tetapi pada sistem tidak terjadi *deadlock*. Terlihat bahwa proses *P*4 kemungkinan melepas tipe sumber daya *R*2. Sumber daya tersebut kemudian dapat dialokasikan untuk *P*3 dan akan menghapus siklus.



Gambar 1-7 *Resource allocation graph* yang tidak terjadi deadlock

2. METODE MENANGANI *DEADLOCK*

Terdapat tiga metode untuk menangani permasalahan *deadlock* yaitu :

- **Metode Pencegahan** : Menggunakan protocol untuk menjamin bahwa sistem tidak pernah memasuki status *deadlock.*
- **Metode Menghindari *deadlock*** : Mengizinkan sistem memasuki status *deadlock* dan kemudian memperbaikinya.
- Mengabaikan permasalahan dan seakan - akan *deadlock* tidak pernah terjadi pada system **(Strategi *Ostrich*).**

3. MENCEGAH *DEADLOCK*

Metode ini berkaitan dengan pengkondisian sistem agar menghilangkan kemungkinan terjadinya *deadlock*. Pencegahan merupakan solusi yang bersih dipandang dari sudut tercegahnya *deadlock*. Metode ini sering menghasilkan utilisasi sumber daya yang buruk. Pencegahan *deadlock* merupakan metode yang banyak dipakai. Untuk mencegah *deadlock* dilakukan dengan meniadakan salah satu dari syarat perlu sebagai berikut :

A. Mencegah Mutual Exclusion

Mutual exclusion adalah hal yang tidak dapat dihindari. Hal ini dikarenakan tidak ada sumber daya yang dapat digunakan bersama-sama, jadi sistem harus membawa sumber daya yang tidak dapat digunakan bersama-sama.

B. Mencegah Hold and Wait

Pencegahan menggunakan *hold and wait* adalah sistem harus menjamin bila suatu proses meminta sumber daya, maka proses tersebut tidak sedang memegang sumber daya yang lain. Proses harus meminta dan semua sumber daya yang diperlukan dialokasikan sebelum proses memulai eksekusi atau mengizinkan proses meminta sumber daya hanya jika proses tidak membawa sumber daya lain. Model ini mempunyai utilitas sumber daya yang rendah dan kemungkinan terjadi *starvation* jika proses membutuhkan sumber daya yang popular sehingga terjadi keadaan menunggu yang tidak terbatas karena

setidaknya satu dari sumber daya yang dibutuhkannya dialokasikan untuk proses yang lain.

C. Mencegah Non - Preemption

Peniadaan *non - preemption* mencegah proses - proses lain harus menunggu. Seluruh proses menjadi *preemption*, sehingga tidak ada tunggu - menunggu. Cara mencegah kondisi *non - preemption* :

o Jika suatu proses yang membawa beberapa sumber daya meminta sumber daya lain yang tidak dapat segera dipenuhi untuk dialokasikan pada proses tersebut, maka semua sumber daya yang sedang dibawa proses tersebut harus dibebaskan.

o Proses yang sedang dalam keadaan menunggu, sumber daya yang dibawanya ditunda dan ditambahkan pada daftar sumber daya.

o Proses akan di restart hanya jika dapat memperoleh sumber daya yang lama dan sumber daya baru yang diminta.

D. Mencegah Kondisi Menunggu Sirkular

Sistem mempunyai total permintaan global untuk semua tipe sumber daya. Proses dapat meminta proses kapanpun diinginkan, tetapi permintaan harus dibuat terurut secara numerik. Setiap proses yang membutuhkan sumber daya dan memintanya maka nomor urut akan dinaikkan. Cara ini tidak akan menimbulkan siklus. Masalah yang timbul adalah tidak ada cara pengurutan nomor sumber daya yang memuaskan semua pihak.

4. MENGHINDARI *DEADLOCK*

Metode alternatif untuk menghindari *deadlock* adalah menggunakan informasi tambahan tentang bagaimana sumber daya diminta. Misalnya, pada sistem dengan satu *tape drive* dan satu printer, proses *P* pertama meminta *tape drive* dan kemudian printer sebelum melepaskan kedua sumber daya tersebut. Sebaliknya proses *Q* pertama meminta printer kemudian *tape drive*. Dengan mengetahui urutan permintaan dan pelepasan sumber daya untuk setiap proses, dapat diputuskan bahwa untuk setiap permintaan apakah proses harus menunggu atau tidak. Setiap permintaan ke sistem harus dipertimbangkan apakah sumber daya tersedia, sumber daya sedang dialokasikan untuk proses dan permintaan kemudian serta pelepasan oleh proses untuk menentukan apakah permintaan dapat dipenuhi atau harus menunggu untuk menghindari *deadlock*.

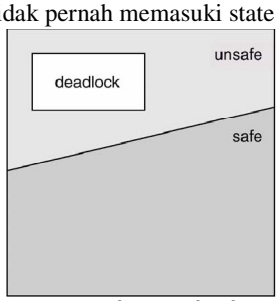
Model yang sederhana dan sangat penting dibutuhkan adalah setiap proses menentukan jumlah maksimum sumber daya dari setiap tipe yang mungkin diperlukan. Algoritma *deadlock avoidance* secara dinamis memeriksa status sumber daya yang dialokasikan untuk menjamin tidak pernah terjadi kondisi menunggu sirkular. Status alokasi sumber daya ditentukan oleh jumlah sumber daya yang tersedia dan yang dialokasikan dan maksimum permintaan oleh proses - proses.

Untuk penghindaran *deadlock* diperlukan pengertian mengenai state selamat (*safe state*) dan state tak selamat (*unsafe state*).

4.1. State Selamat (*Safe State*)

Ketika suatu proses meminta sumber daya yang tersedia, sistem harus menentukan apakah alokasi sumber daya pada proses mengakibatkan sistem dalam state selamat. Sistem dikatakan dalam state selamat jika sistem dapat mengalokasikan sumber daya untuk setiap proses secara berurutan dan menghindari *deadlock*. Urutan proses <*P*1, *P*2, …, *Pn*> selamat jika untuk setiap *P*i, sumber daya yang masih diminta *P*i masih memenuhi sumber daya yang tersedia dan sumber daya yang dibawa oleh setiap *P*j, dimana $j < i$. Jika sumber daya yang diperlukan *P*i tidak dapat segera disediakan, maka *P*i dapat menunggu sampai semua *P*j selesai. Ketika *P*j selesai, *P*i dapat memperoleh sumber daya yang diperlukan, mengeksekusi, mengembalikan sumber daya yang dialokasikan dan terminasi. Ketika *P*i selesai, *P*i +1 dapat memperoleh sumber daya yang diperlukan dan seterusnya.

Jika sistem dalam state selamat maka tidak terjadi *deadlock*, sedangkan jika sistem dalam state tidak selamat (*unsafe state*) maka kemungkinan terjadi *deadlock* seperti Gambar 4-1. Metode menghindari *deadlock* menjamin bahwa sistem tidak pernah memasuki state tidak selamat.



Gambar 4-1 Ruang state selamat, tak selamat, dan *deadlock*

Untuk menggambarkan sistem dapat berpindah dari state selamat ke state tidak selamat dapat dilihat ilustrasi berikut ini. Misalnya, sistem mempunyai 12 *magnetic tape drive* dan 3 proses *P*0, *P*1, dan *P*2. Proses *P*0 membutuhkan 10 *tape drive*, proses *P*1 membutuhkan 4, dan proses *P*2 membutuhkan 9 *tape drive*. Misalnya pada waktu t_0, proses *P*0 membawa 5 *tape drive*, *P*1 membawa 2 dan *P*2 membawa 2 *tape drive* sehingga terdapat 3 *tape drive* yang tidak digunakan.

	Kebutuhan Maksimum	Kebutuhan Sekarang
*P*0	10	5
*P*1	4	2
*P*2	9	2

Pada waktu t_0, sistem dalam state selamat. Urutan <*P*1, *P*0, *P*2> memenuhi kondisi selamat karena *P*1 dapat segera dialokasikan semua *tape drive* dan kemudian mengembalikan semua tape drive sehingga sistem tersedia 5 *tape drive*. Kemudian *P*0 dapat memperoleh semua *tape drive* dan mengembalikan semua sehingga sistem tersedia 10 *tape drive* dan terakhir proses *P*2 dapat

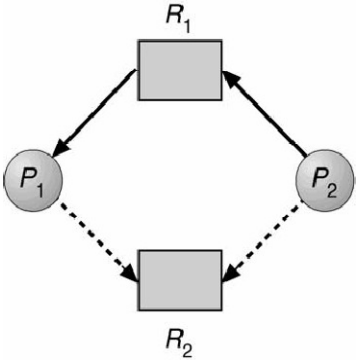
memperoleh semua *tape drive* dan mengembalikan semua *tape drive* sehingga system tersedia 12 *tape drive*.

Sistem dapat berubah dari state selamat ke state tidak selamat. Misalnya, pada waktu t_1, proses *P*2 meminta tambahan alokasi 1 *tape drive*. Sistem menjadi tidak selamat. Pada saat ini, hanya proses *P*1 yang mendapatkan semua *tape drive* dan kemudian mengembalikan semua *tape drive* sehingga hanya tersedia 4 *tape drive*. Karena proses *P*0 sudah dialokasikan 5 *tape drive* tetapi membutuhkan maksimum 10 *tape drive* sehingga meminta 5 *tape drive* lagi. Karena tidak tersedia, proses *P*0 harus menunggu demikian juga *P*2 sehingga system menjadi *deadlock*.

4.2. Algoritma *Resource Allocation Graph*

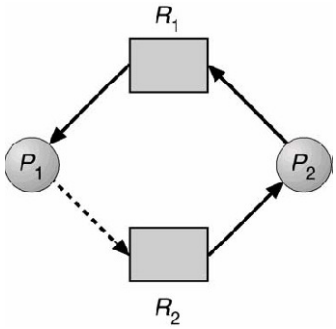
Untuk menghindari *deadlock* pada sistem yang hanya mempunyai satu anggota untuk setiap tipe sumber daya, dapat digunakan algoritma *resource allocation graph*. *Claim edge* *P*i → *R*j menandakan bahwa proses *P*i mungkin meminta sumber daya *R*j yang direpresentasikan dengan garis putus-putus. *Claim edge* akan berubah ke *request edge* bila proses meminta sumber daya. Bila sumber daya dibebaskan oleh proses, *assignment edge* diubah ke *claim edge*. Sumber daya sebelumnya harus diklaim pada sistem. Sehingga sebelum proses *P*i mulai dieksekusi, semua *claim edge* harus muncul pada *resource allocation graph*.

Misalnya, proses Pi meminta sumber daya Rj. Permintaan dapat dipenuhi hanya jika mengubah *request edge* *P*i → *R*j ke *assignment edge* *R*j → *P*i tidak menyebabkan siklus pada graf. Jika tidak terdapat siklus, maka alokasi sumber daya menyebabkan sistem dalam state selamat. Jika terjadi siklus, maka alokasi akan membawa sistem pada state tak selamat. Sehingga proses *P*i harus menunggu permintaan dipenuhi.



Gambar 4-2 Menghindari *deadlock* dengan algoritma *resouce allocation graph*

Untuk menggambarkan algoritma ini, perhatikan *resource allocation graph* Gambar 4-2. Misalnya *P*2 meminta *R*2. Meskipun *R*2 bebas, tetapi tidak dapat dialokasikan untuk *P*2, karena akan menyebabkan siklus pada graf (Gambar 4-3). Siklus menandakan sistem dalam state tak selamat. Jika *P*1 meminta *R*2 dan *P*2 meminta *R*1, maka terjadi *deadlock*.



Gambar 4-3 State tak selamat *pada resouce allocation graph*

4.3. Algoritma Banker

Algoritma *resource allocation graph* tidak dapat diaplikasikan pada sistem yang mempunyai beberapa anggota pada setiap tipe sumber daya. Setiap proses sebelum dieksekusi harus menentukan jumlah sumber daya maksimum yang dibutuhkan. Jika suatu proses meminta sumber daya kemungkinan proses harus menunggu. Jika suatu proses mendapatkan semua sumber daya maka proses harus mengembalikan semua sumber daya dalam jangka waktu tertentu.

Struktur data yang digunakan untuk mengimplementasikan algoritma Banker akan menentukan state dari sumber daya yang dialokasikan oleh sistem. Misalnya, *n* = jumlah proses dan *m* = jumlah tipe *resource*. Struktur data yang diperlukan :

- *Available :* Vektor panjang *m*. Jika *Available*[*j*] = *k*, terdapat *k* anggota tipe sumber daya *R*j yang tersedia.
- *Max :* matrik *n x m*. Jika *Max*[*i, j*] = *k*, maka proses *P*i meminta paling banyak *k* anggota tipe *resource R*j.
- *Allocation :* matriks *n* x *m*. Jika *Allocation*[*i, j*] = *k* maka *P*i sedang dialokasikan *k* anggota tipe *resource R*j.
- *Need :* matriks *n* x *m*. Jika *Need*[*i, j*] = *k*, maka *P*i membutuhkan *k* anggota tipe *resource R*j untuk menyelesaikan tugas bagiannya. *Need*[*i, j*] = *Max*[*i, j*] – *Allocation*[*i, j*].

Beberapa notasi yang perlu diketahui adalah misalnya *X* dan *Y* adalah vektor dengan panjang *n*. $X \leq Y$ jika dan hanya jika $X[i] \leq Y[i]$ untuk semua $i = 1, 2, .., n$. Sebagai contoh jika $X = (1, 7, 3, 2)$ dan $Y = (0, 3, 2, 1)$ maka $Y \leq X$.

4.3.1. Algoritma *Safety*

Algoritma ini untuk menentukan apakah sistem berada dalam state selamat atau tidak.

1. *Work* dan *Finish* adalah vektor dengan panjang *m* dan *n.* Inisialisasi : *Work* = *Available* dan *Finish* [*i*] = false untuk $i = 1,3, \ldots, n$.
2. Cari *i* yang memenuhi kondisi berikut :
   (a) *Finish* [*i*] = false
   (b) *Need* [*i*] ≤ *Work*
   Jika tidak terdapat *i* ke langkah 4.
3. *Work* = *Work* + *Allocation* [*i*]
   Finish[*i*] = true
   Kembali ke langkah 2.
4. Jika *Finish* [*i*] = true untuk semua *i*, maka sistem dalam state selamat.

4.3.2. Algoritma *Resource Request*

Request*i* adalah vektor permintaan untuk proses *P*i. Jika Request$_i$ [*j*] = *k*, maka proses *P*i menginginkan *k* anggota tipe sumber daya *R*j. Jika permintaan untuk sumber daya dilakukan oleh proses *P*i berikut ini algoritmanya. *Request = request* vektor untuk proses *Pi*. **If Request*i* [*j*] = *k* then process *Pi* wants *k* instances of resource type *Rj***.

1. Jika Request*i* ≤ Need*i* ke langkah 2. Selain itu, terjadi kondisi error karena proses melebihi maksimum klaim.
2. Jika Request*i* ≤ *Available*, ke langkah 3. Selain itu *P*i harus menunggu karena sumber daya tidak tersedia.
3. Alokasikan sumber daya untuk *P*i dengan modifikasi state berikut :

   Available = Available - Request*i*;
   Allocation*i* = Allocation*i* + Request*i*;
   Need*i* = Need*i* – Request*i*;

Jika hasil alokasi sumber daya dalam state selamat, maka sumber daya dialokasikan ke *P*i, sebaliknya jika dalam state tidak selamat, *P*i harus menunggu dan state alokasi sumber daya yang lama disimpan kembali.

Contoh Penggunaan Algoritma Banker

Diketahui sistem terdapat 5 proses yaitu *P*0 sampai *P*4, 3 tipe sumber daya yaitu *A* (10 anggota), *B* (5 anggota) dan *C* (7 anggota). Perhatikan gambaran sistem pada waktu T_0.

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Isi matriks *Need* didefinisikan dengan *Max – Allocation*.

	Need		
	A	B	C
*P*0	7	4	3
*P*1	1	2	2
*P*2	6	0	0
*P*3	0	1	1
*P*4	4	3	1

Sistem dalam keadaan state selamat dengan urutan <*P*1, *P*3, *P*4, *P*2, *P*0> yang memenuhi kriteria algoritma *safety*.

Misalnya proses *P*1 meminta tambahan anggota tipe sumber daya *A* dan dua anggota tipe sumber daya *C* sehingga Request*1* = (1, 0, 2). Untuk menentukan apakah permintaan dapat segera dipenuhi, pertama harus diperiksa apakah Request*1* ≤ *Available* ((1, 0, 2) ≤ (3, 3, 2)) ternyata benar. Maka akan diperoleh state baru berikut:

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
*P*0	0	1	0	7	4	3	2	3	0
*P*1	3	0	2	0	2	0			
*P*2	3	0	1	6	0	0			
*P*3	2	1	1	0	1	1			
*P*4	0	0	2	4	3	1			

Kemudian yang harus ditentukan adalah apakah sistem berada dalam state selamat. Setelah mengeksekusi algoritma s*afety* ternyata urutan <*P*1, *P*3, *P*4, *P*0, *P*2> memenuhi kriteria *safety*.

Setelah sistem berada pada state diatas, permintaan (3, 3, 0) oleh *P*4 tidak dapat dipenuhi karena sumber daya tidak tersedia. Permintaan (0, 2, 0) oleh *P*1 juga tidak dapat dipenuhi karena meskipun sumber daya tersedia, hasilnya state tidak selamat.

5. MENDETEKSI *DEADLOCK*

Jika sistem tidak menyediakan algoritma mencegah *deadlock* dan menghindari *deadlock,* maka terjadi *deadlock*. Pada lingkungan ini sistem harus menyediakan:

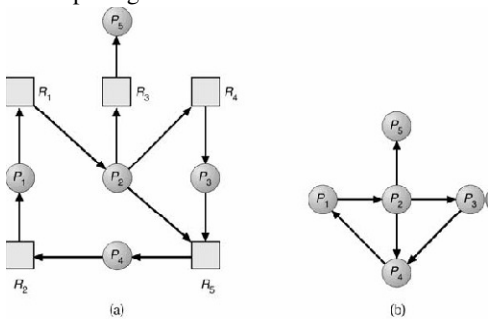
- Algoritma yang menguji state sistem untuk menentukan apakah *deadlock* telah terjadi.
- Algoritma untuk memperbaiki dari *deadlock*.

5.1. Satu Anggota untuk Setiap Tipe Sumber Daya

Jika semua sumber daya hanya mempunyai satu anggota, kita dapat menentukan algoritma mendeteksi *deadlock* menggunakan bentuk *resource allocation graph* yang disebut *wait-for graph*.

Garis dari *P*i → *P*j pada *wait-for graph* menandakan bahwa proses *P*i menunggu *P*j melepaskan sumber daya yang dibutuhkan Pi. Garis *P*i → *P*j terdapat pada *wait-for graph* jika dan anya jika *resource allocation graph* berisi dua garis *P*i → *R*q dan *R*q → *P*j untuk beberapa sumber daya *R*q seperti Gambar 5-1.

Secara periodik sistem menggunakan algoritma yang mencari siklus pada graf. Algoritma untuk mendeteksi siklus pada graf membutuhkan operasi n^2dimana *n* adalah jumlah titik pada graf.



Gambar 5-1 (a) *Resource allocation graph* (b) *Wait-for graph*

5.2. Beberapa Anggota untuk Setiap Tipe Sumber Daya

Untuk tipe sumber daya yang mempunyai beberapa anggota digunakan algoritma yang sejenis dengan

algoritma Banker dengan struktur daya seperti di bawah ini :

- *Available :* vektor panjang *m* menandakan jumlah sumber daya yang tersedia untuk setiap tipe sumber daya.
- *Allocation :* matriks *n* x *m* yang mendefinisikan jumlah sumber daya untuk setiap tipe sumber daya yang sedang dialokasikan untuk setiap proses.
- *Request :* matriks *n* x *m* yang mendefinisikan permintaan setiap proses. Jika *Request* [*I, j*] = *k*, maka proses *P*i meminta *k* anggota tipe sumber daya *R*j.

Algoritma mendeteksi *deadlock* mempunyai urutan berikut :

1. *Work* dan *Finish* adalah vektor panjang *m* dan *n*. Inisialisasi *Work* = *Available*. Untuk *i* = 1, 2, …, *n*, jika Allocation*i* ≠ 0, maka *Finish*[i] = false; sebaliknya *Finish*[i] = true.
2. Cari indeks *i* yang memenuhi kondisi berikut :
   (a) *Finish*[*i*] = false
   (b) Request*i* ≤ *Work*
   Jika tidak terdapat *i* ke langkah 4.
3. *Work* = *Work* + Allocation*i*
   Finish[*i*] = true
   Ke langkah 2.
4. Jika *Finish*[*i*] = false, untuk beberapa *i*, 1 ≤ *i* ≤ *n*, maka sistem berada pada state *deadlock* state. Jika *Finish*[*i*] = false, maka *P*i *deadlock*.

Algoritma ini memerlukan operasi O(*m* x *n*2) untuk mendeteksi apakah sistem berada pada state *deadlock*.

5.3. Penggunaan Algoritma Deteksi

Untuk menjawab kapan dan berapa sering menggunakan algoritma deteksi, hal ini tergantung pada :

- Seberapa sering terjadi *deadlock*.
- Berapa proses yang perlu dilakukan *roll back*.

Jika algoritma deteksi digunakan, terdapat beberapa siklus pada graf, hal ini tidak dapat mengetahui berapa proses yang *deadlock* yang menyebabkan *deadlock*.

6. PERBAIKAN *DEADLOCK*

Terdapat dua pilihan untuk membebaskan deadlock. Satu solusi sederhana adalah dengan menghentikan satu atau beberapa proses untuk membebaskan kondisi menunggu sirkular. Pilihan kedua adalah menunda beberapa sumber daya dari satu atau lebih proses yang *deadlock*.

6.1. Terminasi Proses

Untuk memperbaiki *deadlock* dengan terminasi proses, dapat digunakan salah satu dari dua metode di bawah ini :

- Menghentikan (*abort*) semua proses yang *deadlock*
- Menghentikan satu proses setiap waktu sampai siklus *deadlock* hilang.

Untuk menentukan urutan proses yang harus dihentikan ada beberapa faktor yang harus diperhatikan :

- Prioritas proses.
- Berapa lama proses dijalankan dan berapa lama lagi selesai ?
- Sumber daya yang digunakan selama proses.
- Sumber daya yang diperlukan proses untuk menyelesaikan tugas dan mencapai tujuan.
- Berapa proses yang perlu diterminasi ?
- Apakah proses bersifat interaktif atau *batch* ?

6.2. Menunda Sumber Daya

Untuk menghilangkan *deadlock* dengan menunda sumber daya, sumber daya dari proses harus ditunda dan memberikan sumber daya tersebut ke proses lain sampai siklus *deadlock* hilang.

Jika penundaan dibutuhkan untuk menghilangkan *deadlock*, terdapat tiga hal yang perlu diperhatikan :

- Pilihlah korban (sumber daya) yang mempunyai biaya minimal.
- Lakukan *rollback* yaitu memulai kembali (*restart*) proses pada state yang selamat.
- Harus dijamin *starvation* tidak akan terjadi karena kemungkinan beberapa proses selalu terpilih sebagai korban termasuk jumlah *rollback* sebagai faktor biaya.

7. METODE KOMBINASI MENANGANI *DEADLOCK*

Untuk menangani *deadlock* dilakukan kombinasi dari tiga algoritma dasar yaitu mencegah *deadlock*, menghindari *deadlock* dan mendeteksi *deadlock*. Kombinasi ketiga algoritma ini memungkinkan penggunaan yang optimal untuk setiap sumber daya pada sistem.

8. KESIMPULAN

Deadlock adalah keadaan pada proses yang berhenti akibat keadaan saling menunggu. *Deadlock* terjadi karena *mutual exclusion, hold and wait, non – preemption,* menunggu secara sirkuler. Masalah ini dapat diselesaikan dengan metode pencegahan, metode menghindari, metode Ostrich. Jika sistem tidak menyediakan algoritma mencegah ataupun meghindari *deadlock*, maka *deadlock* harus dideteksi dan diperbaiki. Tantangan pada distribusi data adalah menghindari masalah *deadlock*, dimana banyak klien mengakses data secara bersamaan dan data harus tetap konsisten (konkurensi).

REFERENSI

[1] http://en.wikipedia.org/wiki/Deadlock diakses 7/12/2010 4:45 PM

[2] http://lecturer.eepis-its.edu/~arna/Diktat_SO/6.Deadlock.pdf diakses 7/12/2010 4:45 PM

[3] Munir, Rinaldi. 2009. Matematika Diskrit. Bandung: Informatika.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 8 Desember 2010

ttd

Dion Jogi Parlinggoman
13509045