

Bab 6

Deadlock

POKOK BAHASAN:

- ✓ Model Sistem
- ✓ Karakteristik Deadlock
- ✓ Metode untuk Menangani Deadlock
- ✓ Mencegah Deadlock
- ✓ Menghindari Deadlock
- ✓ Mendeteksi Deadlock
- ✓ Perbaikan dari Deadlock
- ✓ Kombinasi Penanganan Deadlock

TUJUAN BELAJAR:

Setelah mempelajari materi dalam bab ini, mahasiswa diharapkan mampu:

- ✓ Memahami latar belakang terjadinya deadlock pada sistem.
- ✓ Memahami karakteristik bagaimana deadlock bisa terjadi.
- ✓ Memahami metode untuk menangani deadlock yang meliputi mencegah deadlock, menghindari deadlock, mendeteksi deadlock dan perbaikan dari 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.

Misalnya sistem mempunyai 2 tape drive dan terdapat dua proses P_1 dan P_2 yang masing masing membawa satu tape drive dan masing-masing memerlukan tape drive yang dibawa proses lain sehingga terjadi keadaan saling menunggu resource dan sistem di-blok.

Contoh lain, misalnya terdapat semaphore A dan B yang diinisialisasi 1 dan terdapat dua proses P_0 dan P_1 masing-masing membawa semaphore A dan B . Kemudian P_0 dan P_1 meminta semaphore B dan A dengan menjalankan operasi *wait*. Hal ini mengakibatkan proses di-blok dan terjadi deadlock.

P_0	P_1
<i>wait</i> (A);	<i>wait</i> (B);
<i>wait</i> (B);	<i>wait</i> (A);

6.1 MODEL SISTEM

Pada sistem terdapat beberapa sumber daya (*resource*) yang digunakan untuk proses-proses untuk menyelesaikan task. Sumber daya yang pada sistem terdiri dari tipe resource CPU cycle, ruang memori, perangkat I/O yang disebut dengan tipe sumber daya R_1, R_2, \dots, R_m . Setiap tipe sumber daya R_i mempunyai beberapa anggota W_i . 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

6.2 KARAKTERISTIK DEADLOCK

6.2.1 Kondisi yang Diperlukan

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 menunggu mendapatkan tambahan sumber daya baru yang dibawa oleh proses

- c. **Non-Preemption** : sebuah sumber daya dapat dibebaskan dengan sukarela oleh proses yang memegangnya setelah proses menyelesaikan task.
- d. **Menunggu Secara Sirkuler (Circular Wait)** : Terdapat sekumpulan proses $\{P_0, P_1, \dots, P_n\}$ 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, P_{n-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 berarti terpenuhi kondisi-kondisi diatas, tak mungkin terjadi *deadlock* bila tidak ada ketiga kondisi itu. *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.

6.2.2 Resource Allocation Graph

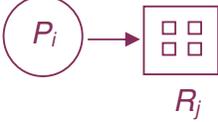
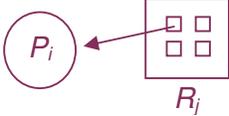
Deadlock dapat digambarkan lebih presisi dengan menggunakan graph berarah yang disebut *resource allocation graph*. Graph terdiri dari himpunan titik V dan garis E . Himpunan titik (vertex) V dibagi menjadi dua tipe yaitu himpunan proses yang aktif pada sistem $P = \{P_1, P_2, \dots, P_n\}$ dan tipe sumber daya pada sistem $R = \{R_1, R_2, \dots, R_m\}$

Garis berarah dari proses P_i ke tipe sumber daya R_j dinotasikan dengan $P_i \rightarrow R_j$ 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 $R_j \rightarrow P_i$ artinya satu anggota tipe sumber daya R_j dialokasikan ke proses P_i . Garis berarah $P_i \rightarrow R_j$ disebut *request edge* dan garis berarah $R_j \rightarrow P_i$ 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 

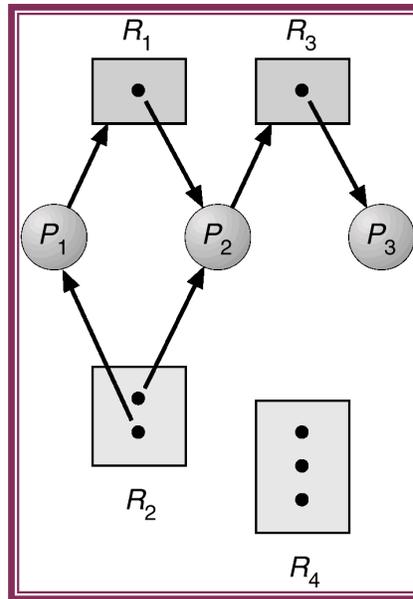
Contoh *resource allocation graph* dapat dilihat pada Gambar 6-1 dimana keadaan sistem adalah sebagai berikut :

- Himpunan P , R dan E :
 - $P = \{P_1, P_2, P_3\}$
 - $R = \{R_1, R_2, R_3, R_4\}$
 - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- 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 graph tidak terdapat siklus maka tidak ada proses dalam sistem yang deadlock

- Apabila pada graph 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



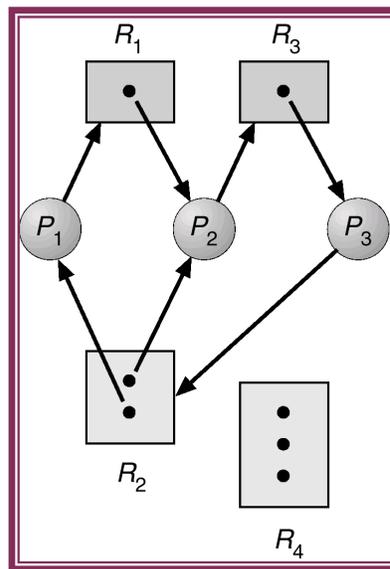
Gambar 6-1 : Contoh Resource Allocation Graph

Untuk ilustrasi konsep diatas kita lihat kembali *resource allocation graph* pada Gambar 6-1. Pada Gambar 6-1 tidak terdapat siklus, jadi tidak terjadi deadlock pada sistem. Misalnya proses P_3 meminta satu anggota dari tipe sumber daya R_2 . Karena tidak tersedia anggota tipe sumber daya tersebut, *request edge* $P_3 \rightarrow R_2$ ditambahkan ke graph seperti pada Gambar 6-2. Pada kasus ini, terdapat dua siklus pada sistem, yaitu :

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

Proses P_1 , P_2 dan P_3 terjadi deadlock. Proses P_2 menunggu sumber daya R_3 yang dibawa proses P_3 . Proses P_3 sebaliknya menunggu proses P_1 atau P_2 melepas sumber daya R_2 . Proses P_1 menunggu proses P_2 melepas sumber daya R_1 .

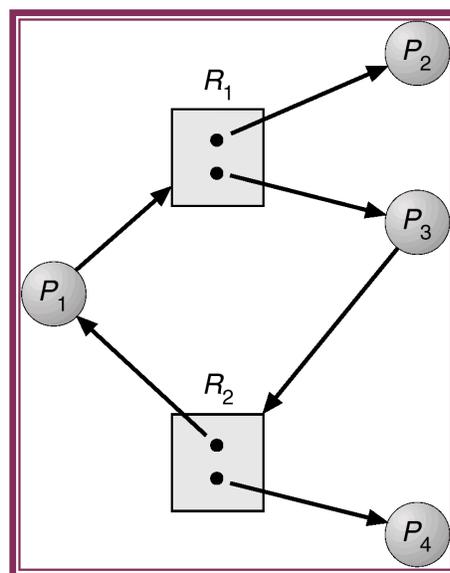


Gambar 6-2 : Resource allocation graph yang terjadi deadlock

Pada contoh *resource allocation graph* Gambar 6-3 terdapat siklus :

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_3 \rightarrow P_1$$

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 6-3 : Resource allocation graph yang tidak terjadi deadlock

6.3 METODE MENANGANI DEADLOCK

Terdapat tiga metode untuk menangani permasalahan deadlock yaitu :

- Menggunakan protocol untuk menjamin bahwa sistem tidak pernah memasuki status deadlock
- Mengizinkan sistem memasuki status deadlock dan kemudian memperbaikinya.
- Mengabaikan permasalahan dan seakan-akan deadlock tidak pernah terjadi pada sistem. Model ini yang banyak digunakan pada sistem operasi termasuk UNIX.

6.4 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 :

- **Mencegah Mutual Exclusion**

Mutual exclusion benar-benar tak 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.

- **Mencegah Hold and Wait**

Untuk mencegah hold and wait, sistem harus menjamin bila suatu proses meminta sumber daya, maka proses tersebut tidak sedang memegang sumber daya yang lain. Proses harus meminta dan dialokasikan semua sumber daya yang diperlukan 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 populer sehingga terjadi keadaan menunggu yang

tidak terbatas karena setidaknya satu dari sumber daya yang dibutuhkananya dialokasikan untuk proses yang lain.

- **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* :

- 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.
- Proses yang sedang dalam keadaan menunggu, sumber daya yang dibawanya ditunda dan ditambahkan pada daftar sumber daya.
- Proses akan di restart hanya jika dapat memperoleh sumber daya yang lama dan sumber daya baru yang diminta.

- **Mencegah Kondisi Menunggu Sirkular**

Sistem mempunyai total permintaan global untuk semua tipe sumber daya. Proses dapat meminta proses kapanpun menginginkan, tapi 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.

6.5 MENGHINDARI DEADLOCK

Metode alternatif untuk menghindari deadlock adalah digunakan 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.

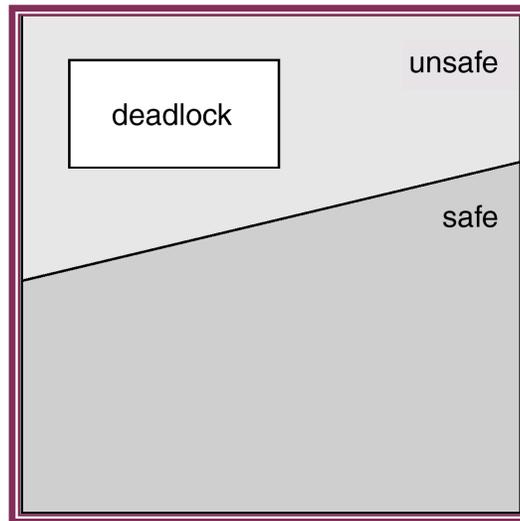
Model yang sederhana dan sangat penting dibutuhkan adalah setiap proses menentukan jumlah maksimum sumber daya dari setiap tipe yang mungkin diperlukan. Algoritma *deadlock avoidance* secara dinamis memeriksa status sumber daya yang dialokasikan untuk menjamin tidak pernah terjadi kondisi menunggu sirkular. 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 aman (*safe state*) dan state tak aman (*unsafe state*).

6.5.1 State Aman (Safe State)

Ketika suatu proses meminta sumber daya yang tersedia, sistem harus menentukan apakah alokasi sumber daya pada proses mengakibatkan sistem dalam state aman. Sistem dikatakan dalam state aman jika sistem dapat mengalokasikan sumber daya untuk setiap proses secara berurutan dan menghindari deadlock. Urutan proses $\langle P_1, P_2, \dots, P_n \rangle$ aman 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 aman maka tidak terjadi deadlock, sedangkan jika sistem dalam state tidak aman (*unsafe state*) maka kemungkinan terjadi deadlock seperti Gambar 6-4. Metode menghindari deadlock menjamin bahwa sistem tidak pernah memasuki state tidak aman.



Gambar 6-4 : Ruang state aman, tak aman dan deadlock

Untuk menggambarkan sistem dapat berpindah dari state aman ke state tidak aman 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.

	<u>Kebutuhan Maksimum</u>	<u>Kebutuhan Sekarang</u>
P_0	10	5
P_1	4	2
P_2	9	2

Pada waktu t_0 , sistem dalam state aman. Urutan $\langle P_1, P_0, P_2 \rangle$ memenuhi kondisi aman 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.

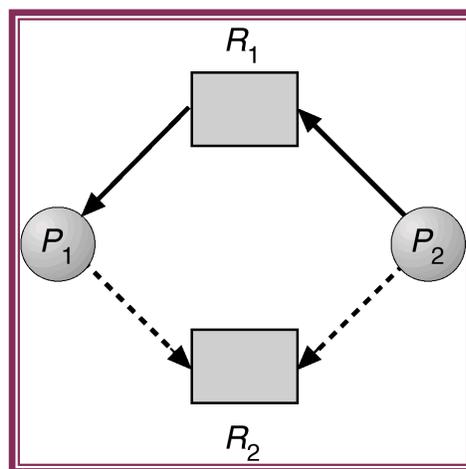
Sistem dapat berubah dari state aman ke state tidak aman. Misalnya pada waktu t_1 , proses P_2 meminta tambahan alokasi 1 tape drive. Sistem menjadi tidak aman. Pada saat ini, hanya proses P_1 yang mendapatkan semua tape drive dan kemudian

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.

6.5.2 Algoritma Resource Allocation Graph

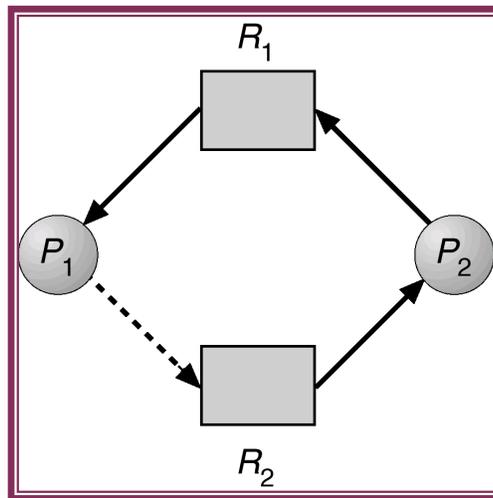
Untuk menghindari deadlock pada sistem yang hanya mempunyai satu anggota untuk setiap tipe sumber daya, dapat digunakan algoritma *resource allocation graph*. *Claim edge* $P_i \rightarrow R_j$ menandakan bahwa proses P_i mungkin meminta sumber daya R_j yang direpresentasikan dengan garis putus-putus. *Claim edge* akan berubah ke *request edge* bila proses meminta sumber daya. Bila sumber daya dibebaskan oleh proses, *assignment edge* $R_j \rightarrow P_i$ pada sistem. Sehingga sebelum proses P_i mulai dieksekusi, semua *claim edge* harus muncul pada *resource allocation graph*.

Misalnya proses P_i meminta sumber daya R_j . Permintaan dapat dipenuhi hanya jika mengubah *request edge* $P_i \rightarrow R_j$ ke *assignment edge* $R_j \rightarrow P_i$ tidak menyebabkan siklus pada graph. Jika tidak terdapat siklus, maka alokasi sumber daya menyebabkan sistem dalam state aman. Jika terjadi siklus, maka alokasi akan membawa sistem pada state tak aman. Sehingga proses P_i harus menunggu permintaan dipenuhi.



Gambar 6-5 : Menghindari deadlock dengan algoritma resource allocation graph

Untuk menggambarkan algoritma ini, perhatikan *resource allocation graph* Gambar 6-5. Misalnya P_2 meminta R_2 . Meskipun R_2 bebas, tetapi tidak dapat dialokasikan untuk P_2 , karena akan menyebabkan siklus pada graph (Gambar 6-6). Siklus menandakan sistem dalam state tak aman. Jika P_1 meminta R_2 dan P_2 meminta R_1 , maka terjadi deadlock.



Gambar 6-6 : State tak aman pada resource allocation graph

6.5.3 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 $n \times m$. Jika $Max[i, j] = k$, maka proses P_i meminta paling banyak k anggota tipe resource R_j .

- *Allocation* : matrik $n \times m$. Jika $Allocation[i, j] = k$ maka P_i sedang dialokasikan k anggota tipe resource R_j .
- *Need* : matrik $n \times m$. Jika $Need[i, j] = k$, maka P_i membutuhkan k anggota tipe resource R_j untuk menyelesaikan task. $Need[i, j] = Max[i, j] - Allocation[i, j]$.

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 $i = 1, 2, \dots, n$. Sebagai contoh jika $X = (1, 7, 3, 2)$ dan $Y = (0, 3, 2, 1)$ maka $Y \leq X$.

6.5.3.1 Algoritma Safety

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

1. *Work* dan *Finish* adalah vector dengan panjang m dan n . Inisialisasi : $Work = Available$ dan $Finish[i] = false$ untuk $i = 1, 3, \dots, n$.
2. Cari i yang memenuhi kondisi berikut :
 - (a) $Finish [i] = false$
 - (b) $Need_i \leq Work$
 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 aman.

6.5.3.2 Algoritma Resource Request

$Request_i$ adalah vector 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 vector for process P_i . If $Request_i [j] = k$ then process P_i wants k instances of resource type R_j .

1. Jika $Request_i \leq Need_i$ ke langkah 2. Selain itu, terjadi kondisi error karena proses melebihi maksimum klaim.

2. Jika $Request_i \leq Available$, 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 state alokasi sumber daya adalah aman, maka sumber daya dialokasikan ke P_i , sebaliknya jika tidak aman, P_i harus menunggu dan state alokasi sumber daya yang lama disimpan kembali.

6.5.3.3 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 .

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	$A \ B \ C$	$A \ B \ C$	$A \ B \ C$
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 matrik *Need* didefinisikan dengan $Max - Allocation$.

	<u>Need</u>
	$A \ B \ C$
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 aman dengan urutan $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ 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 $Request_1 \leq Available$ ($(1, 0, 2) \leq (3, 3, 2)$) ternyata benar. Maka akan diperoleh state baru berikut :

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	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 harus ditentukan apakah sistem berada dalam state aman. Setelah mengeksekusi algoritma safety ternyata urutan $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ memenuhi criteria safety.

Setelah sistem berada pada state doatas, 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, state hasil tak aman.

6.6 MENDETEKSI DEADLOCK

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

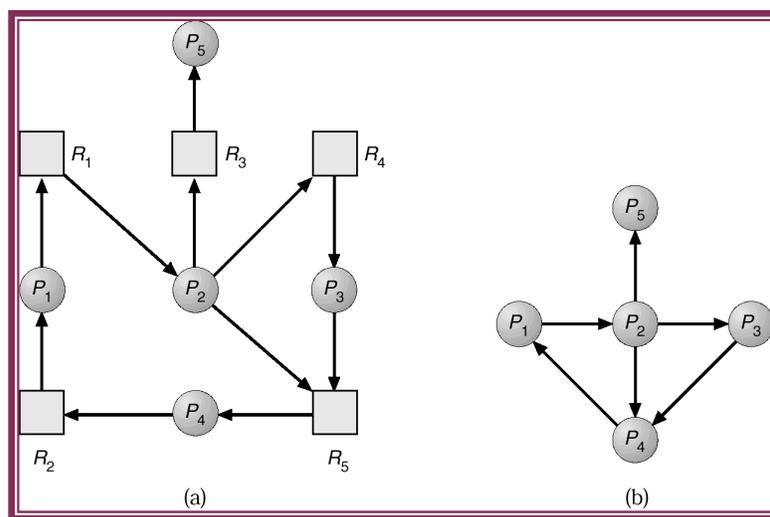
- Algoritma yang menguji state sistem untuk menentukan apakah deadlock telah terjadi.
- Algoritma untuk memperbaiki dari deadlock.

6.6.1 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 $P_i \rightarrow P_j$ pada *wait-for graph* menandakan bahwa proses P_i menunggu P_j melepaskan sumber daya yang dibutuhkan P_i . Garis $P_i \rightarrow P_j$ terdapat pada *wait-for graph* jika dan hanya jika *resource allocation graph* berisi dua garis $P_i \rightarrow R_q$ dan $R_q \rightarrow P_j$ untuk beberapa sumber daya R_q seperti Gambar 6-7.

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



Gambar 6-7 : (a) Resource allocation graph (b) Wait-for graph

6.6.2 Beberapa Anggota untuk Setiap Tipe Sumber Daya

Untuk Tipe sumber daya yang mempunyai beberapa anggota digunakan algoritma yang sejenis dengan algoritma Banker dengan struktur data seperti di bawah ini :

- *Available* : vector panjang m menandakan jumlah sumber daya yang tersedia untuk setiap tipe sumber daya.
- *Allocation* : matrik $n \times m$ yang mendefinisikan jumlah sumber daya untuk setiap tipe sumber daya yang sedang dialokasikan untuk setiap proses.
- *Request* : matrik $n \times m$ 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 $i = 1, 2, \dots, n$, jika $Allocation_i \neq 0$, maka $Finish[i] = false$; sebaliknya $Finish[i] = true$.
2. Cari indeks i yang memenuhi kondisi berikut :
 - (a) $Finish[i] == false$
 - (b) $Request_i \leq Work$
 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 , $1 \leq i \leq n$, maka sistem berada pada state deadlock state. Jika $Finish[i] == true$, maka P_i deadlock

Algoritma ini memerlukan operasi $O(m \times n^2)$ untuk mendeteksi apakah sistem berada pada state deadlock.

Untuk menggambarkan algoritma deteksi, misalnya sistem terdapat 5 proses P_0 sampai P_4 dan 3 tipe sumber daya A , B dan C . Tipe sumber daya A mempunyai 7 anggota, tipe sumber daya B mempunyai 2 anggota dan tipe sumber daya C mempunyai 6 anggota. Pada waktu T_0 , state sumber daya yang dialokasikan adalah :

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	$A \ B \ C$	$A \ B \ C$	$A \ B \ C$
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Sistem tidak berada pada state deadlock karena urutan $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ menghasilkan $Finish[i] = true$ untuk semua i .

Misalnya saat ini proses P_2 membutuhkan tambahan satu anggota tipe sumber daya C . Matrik $Request$ dimodifikasi sebagai berikut :

	<u>Request</u>		
	A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

Sistem sekarang berada pada state deadlock. Meskipun proses P_0 dapat membawa sumber daya, jumlah sumber daya yang tersedia tidak dapat memenuhi permintaan proses lain. Sehingga terjadi deadlock pada proses P_1, P_2, P_3 dan P_4 .

6.6.3 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 graph, hal ini tidak dapat mengetahui berapa proses yang deadlock yang menyebabkan deadlock.

6.7 PERBAIKAN DARI 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.

6.7.1 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 proses.
- Sumber daya proses yang diperlukan untuk menyelesaikan task.
- Berapa proses yang perlu diterminasi.
- Apakah proses interaktif atau batch.

6.7.2 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 aman.
- Harus dijamin *starvation* tidak akan terjadi karena kemungkinan beberapa proses selalu terpilih sebagai korban termasuk jumlah *rollback* sebagai faktor biaya.

6.8 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.

LATIHAN SOAL :

1. Apa yang dimaksud dengan sumber daya ? Berikan contohnya.