Sistem Operasi
7

Deadlock

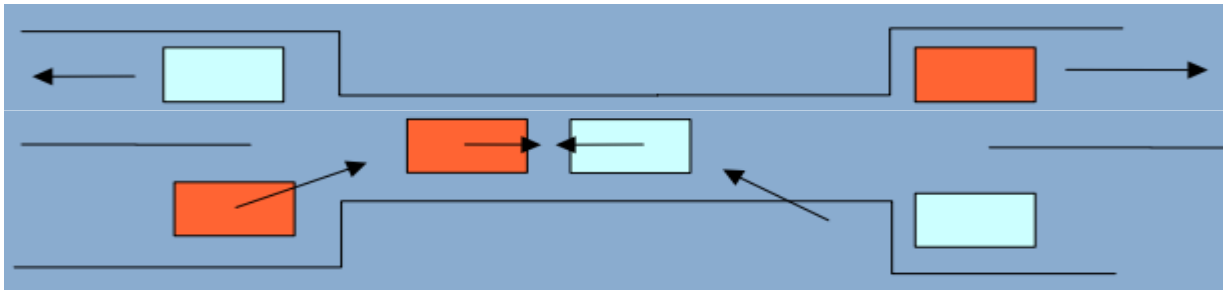
Antonius Rachmat C, S.Kom, M.Cs

Not Responding - Deadlock

- When OS asks a program to do something, like take *a keystroke or close itself*, and the program **fails** to acknowledge that request within a certain amount of time, the program is "**Not Responding**"
- If the program never comes out of that state, we might also call it "**hung**", as in "**hung up**" on something -> **deadlock**

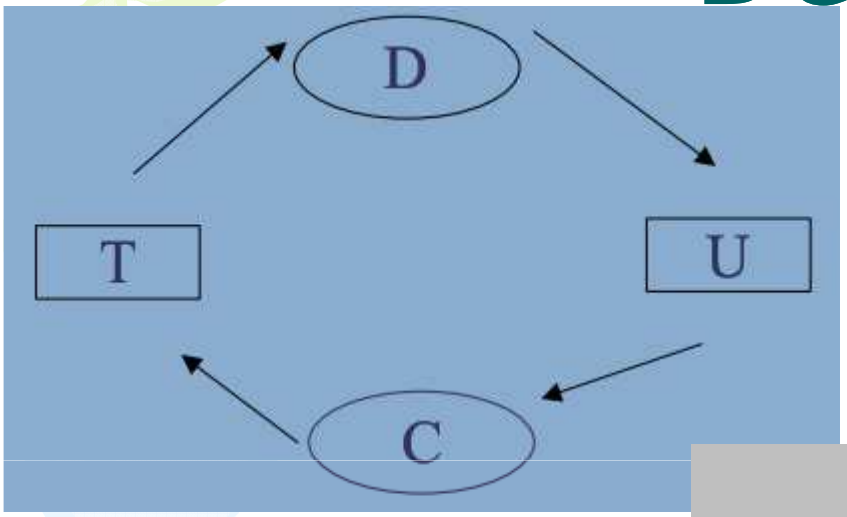
Deadlock

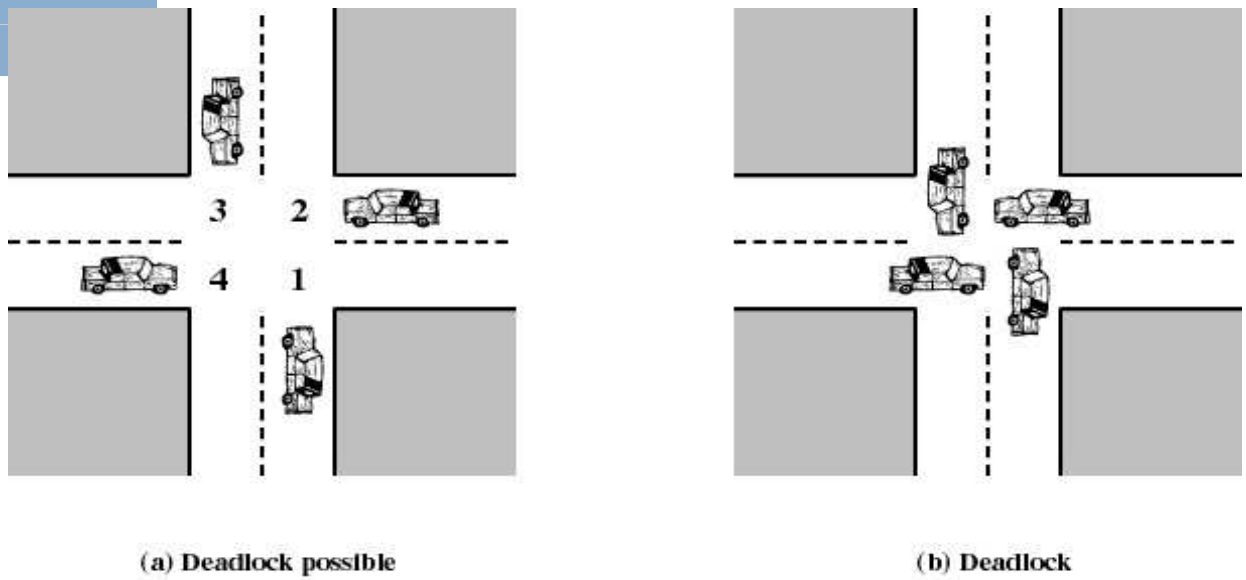
- Jika proses 1 sedang menggunakan sumber daya 1 dan menunggu sumber daya 2 yang ia butuhkan, sedangkan proses 2 sedang menggunakan sumber daya 2 dan menunggu sumber daya 1
- Atau dengan kata lain saat proses masuk dalam status **menunggu**, ia tidak akan pernah selesai menunggu sebab sumber daya yang dibutuhkan sedang digunakan oleh proses lain yang sedang menunggu pula



- Pada satu jalan yang memungkinkan hanya satu arah yang berjalan
- Setiap jalan bisa dianggap sebagai sumber daya
- Saat deadlock terjadi hanya bisa diatasi jika salah satu mobil mundur, dalam hal ini butuh sumber daya yang direalokasikan
- Bahkan beberapa mobil harus mundur jika deadlock terjadi
- Pada kasus ini juga bisa terjadi “starvation”, yaitu ada proses yang tidak terlayani

Deadlock





(a) Deadlock possible

(b) Deadlock

Figure 6.1 Illustration of Deadlock

System Model

- **Resource** types R_1, R_2, . . ., R_m
  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i **instances**.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Jenis Resource

- **Preempt able**: masih boleh diambil dari proses yg sedang memakainya tanpa memberi efek apapun pada proses tersebut
  - Contoh: memory
- **Non preempt able**: tidak boleh diambil dari proses yg sdg memakainya.
  - Contoh: printer
  - Ada **potensi** menyebabkan deadlock

Penyebab Deadlock

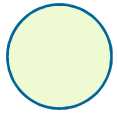
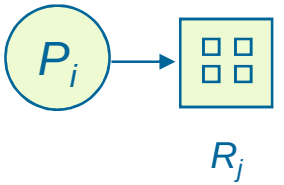
- **Mutual Exclusion**: satu proses satu sumber daya
- **Hold and Wait**: proses yang memegang sumber daya masih bisa meminta sumber daya lain
- **No Preemption**: sumber daya yang sedang digunakan oleh suatu proses tidak bisa sembarangan diambil dari proses tersebut, melainkan harus dilepaskan dengan sendirinya oleh proses.
- **Circular Wait**: setiap proses menunggu sumber daya dari proses berikutnya yg sedang dipakai oleh proses lain.

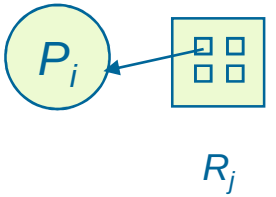
Resource-Allocation Graph

himpunan vertices *V* dan himpunan edges *E*.

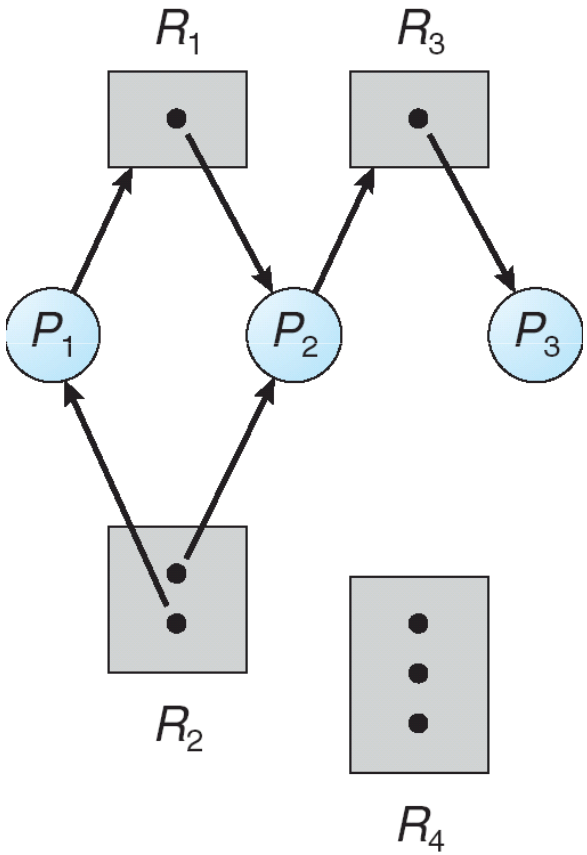
- V dipartisi kedalam 2 tipe:
  - $P = \{P_1, P_2, \ldots, P_n\}$, kumpulan **proses**.
  - $R = \{R_1, R_2, \ldots, R_m\}$, kumpulan **resource**.
- **request** edge – directed edge $P_1 \rightarrow R_j$
- **assignment** edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph

- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

- P_i is holding an instance of R_j 

Example of a Resource Allocation Graph

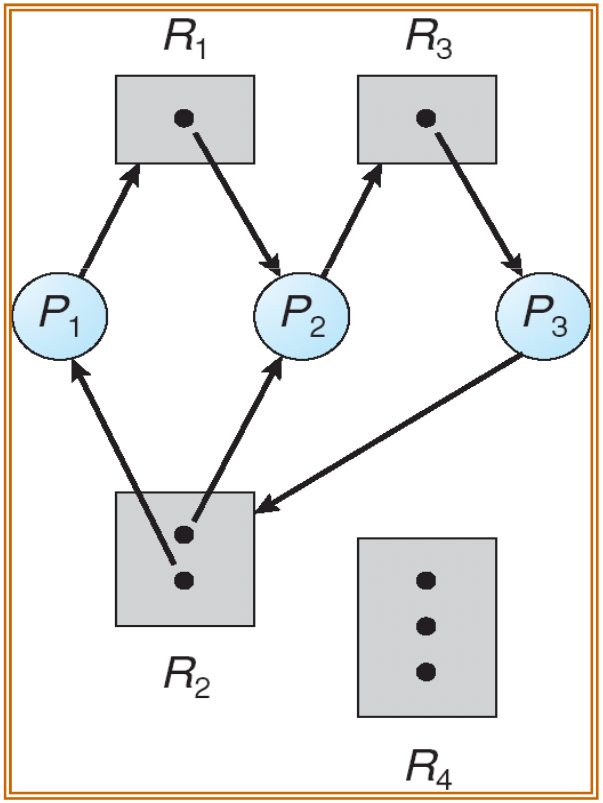


Example of a Resource Allocation Graph with deadlock

There is **closed-cycle**
P1 -> R1 -> P2 -> R3 -> P3 -> R2 -> P1
P2 -> R3 -> P3 -> R2 -> P2

P1, P2, dan P3 akan **deadlock**

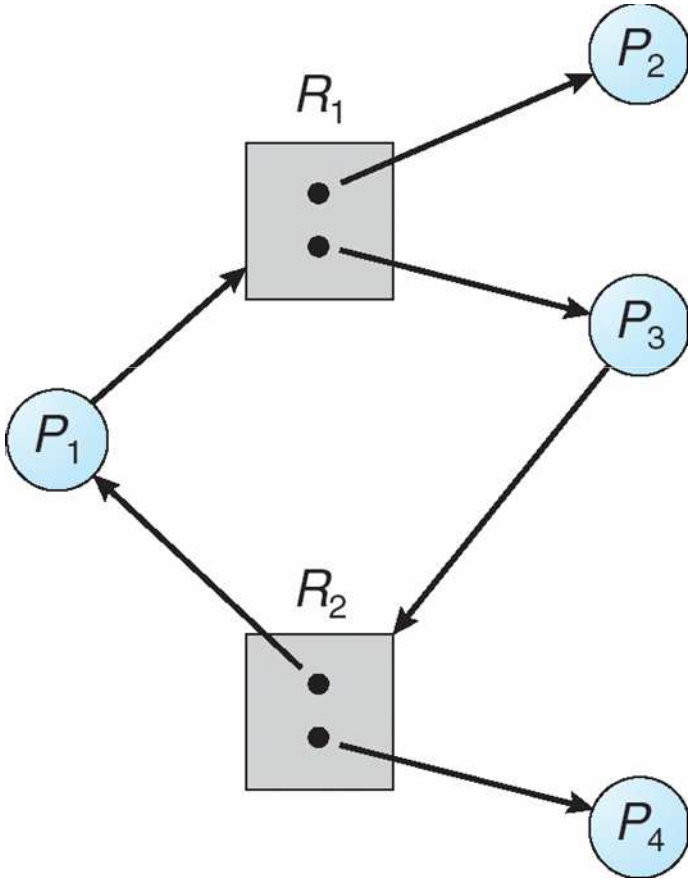
1. P2 menunggu R3 yg dibawa P3
2. P3 menunggu P1 atau P2 untuk membebaskan R2
3. P1 menunggu P2 untuk melepaskan R1



Graph With A Cycle But No Deadlock

There is **closed-cycle**
P1 -> R1 -> P3 -> R2 -> P1

P4 dapat melepaskan R2 dan
kemudian R2 akan dialokasikan ke P3



Basic Facts 1

- If graph contains no cycles $\Rightarrow$ **no deadlock**.

- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then **deadlock**.
  - if several instances per resource type, **possibility** of deadlock.

Handling Deadlock

- Memakai **protokol** untuk memastikan tidak akan pernah mengalami deadlock
  - **Deadlock Avoidance** => memerintahkan pada sistem operasi untuk memberi informasi/prediksi tentang operasi mana yang bisa dan perlu dilaksanakan (keadaan aman).
  - **Deadlock Prevention** => memastikan bahwa keadaan yang penting tidak bisa menunggu
- **Membiarkan** sistem memasuki waktu deadlock, mendeteksinya, dan memperbaikinya
  - Algoritma mendeteksi deadlock
  - Algoritma memperbaiki deadlock
- **Mengabaikan** masalah deadlock, dan menganggap bahwa deadlock tidak akan pernah terjadi lagi

Pencegahan Deadlock (Deadlock Avoidance)

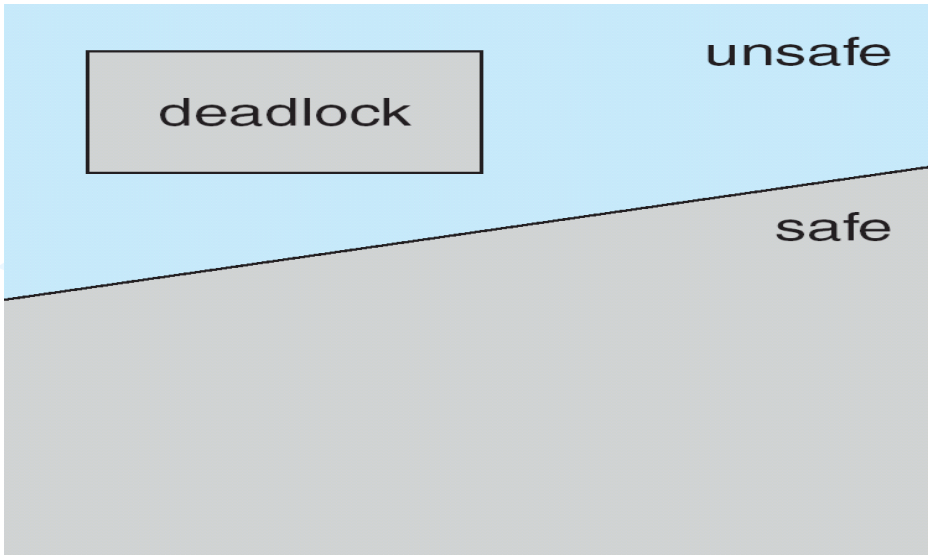
- Caranya adalah **mencegah** adanya deadlock dengan memberikan informasi tambahan ttg resource yg dibutuhkan
  - Banyaknya resource yg tersedia
  - Banyaknya resource yg dialokasikan
  - Banyaknya resource yg dibutuhkan
  - Dan maksimum resource yg dibutuhkan proses
- Menggunakan algoritma prediksi

Basic Facts 2

- If a system is in **safe state** ⇒ **no deadlocks**
- If a system is in **unsafe state** ⇒ **possibility of deadlock**
- **Avoidance** ⇒ ensure that a system **will never** enter an unsafe state.

Safe State

- Saat sistem meminta izin untuk mengambil sumber dayanya, sistem operasi harus memastikan bahwa ia dalam kondisi **aman**
- Sistem dalam kondisi aman jika seluruh sistem dapat berjalan tanpa **terancam** kekurangan sumber daya atau deadlock



Deadlock Avoidance algorithms

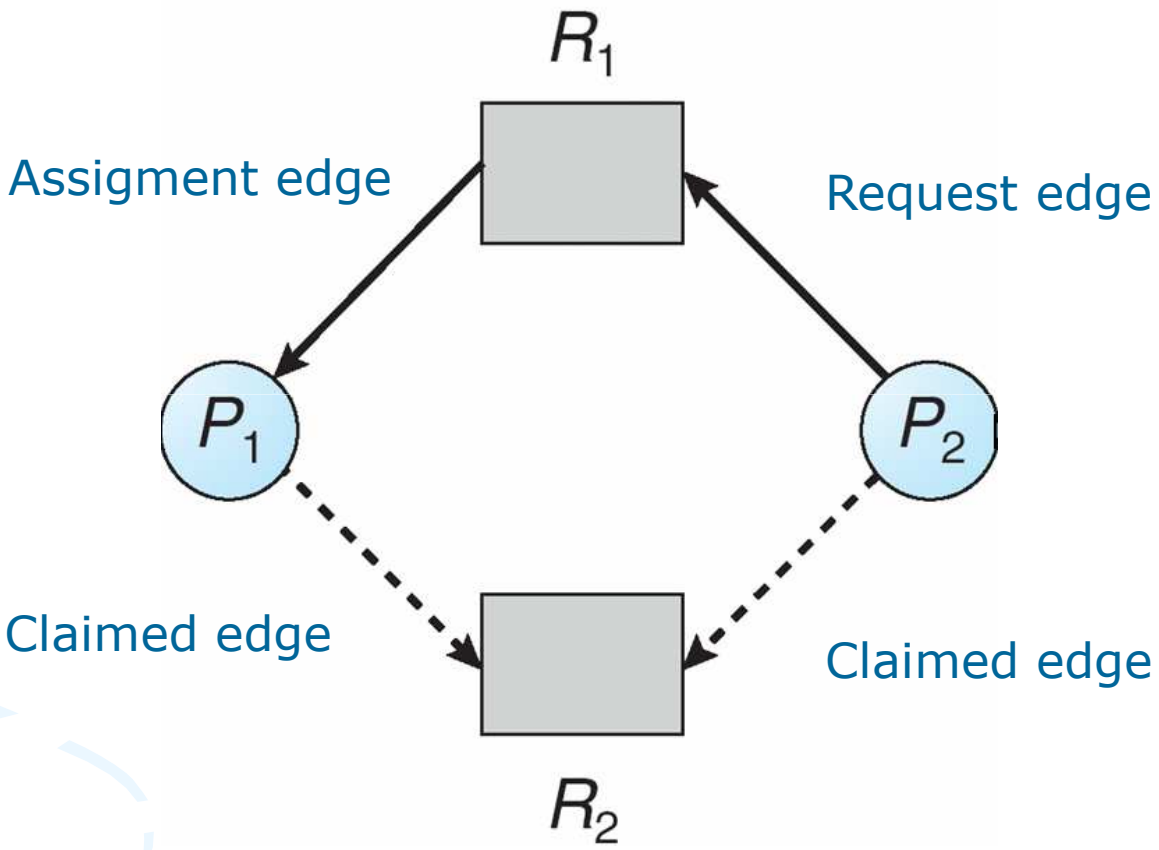
- Single instance of a resource type
  - Use a **resource-allocation graph**
  - Use **wait-for-graph**

- Multiple instances of a resource type
  - Use the **banker's** algorithm

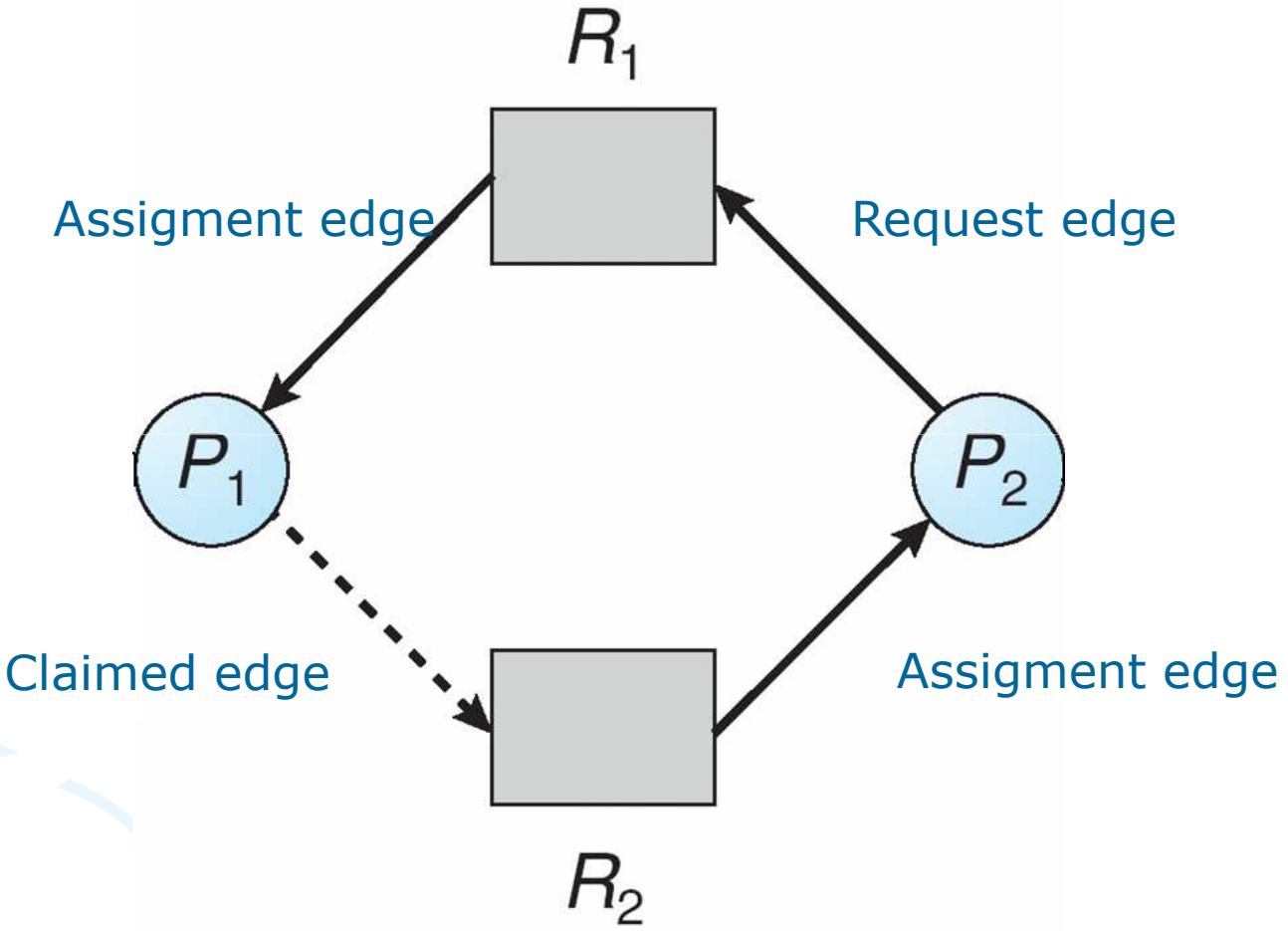
Resource-Allocation Graph Scheme

- **Claimed Edge** Pi → Rj yang menggambarkan ada proses Pi yg **mungkin** akan meminta sumber daya Rj;
  - Mirip dgn request edge tapi dgn garis putus-putus.
- **Claimed Egde** diubah menjadi **Request Edge** ketika ada proses yang memerlukan sumber daya.
- Ketika suatu sumber daya dilepaskan oleh suatu proses, **assigment edge** akan diubah menjadi **claimed edge** lagi.

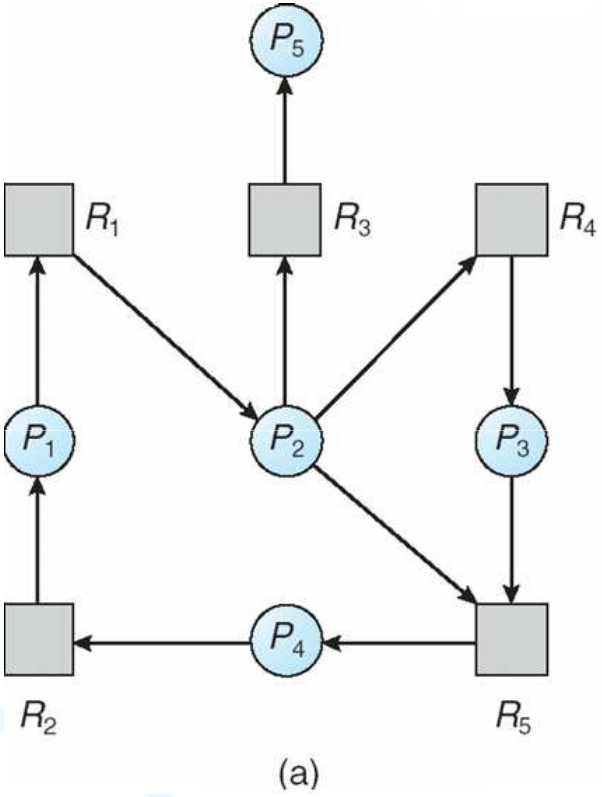
Resource-Allocation Graph

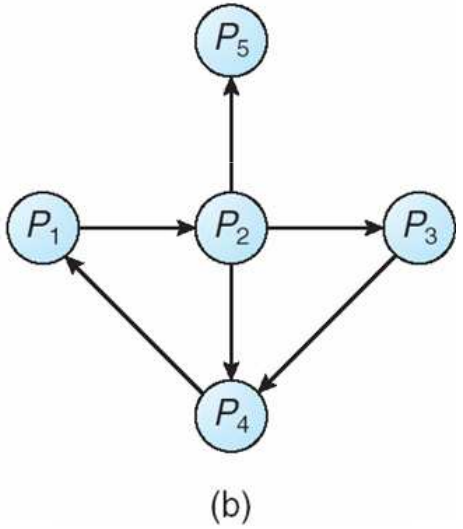


Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph and Wait-for Graph





Resource-Allocation Graph

Corresponding wait-for graph

Banker's Algorithm

- Support **multiple instances**.
- When a process requests a resource it may have to **wait**.
- Setiap proses yang masuk harus memberitahu berapa banyak sumber daya **maksimum** yang dibutuhkan
- Setelah itu sistem mendeteksi apakah sumber daya yang dibutuhkan memang bisa dijalankan dalam kondisi **aman**
  - Jika ya, maka sistem akan melepaskan sumber dayanya untuk digunakan
  - Jika tidak, maka proses harus menunggu hingga sumber dayanya cukup

Data Structures for the Banker's Algorithm

Let *n* = jumlah proses, and *m* = jumlah resource

- **Available***:* Vector of length *m* yg menunjukkan resource-resource yg **tersedia**
- **Max***: n x m* matrix. Yg mendefinisikan **maksimum permintaan** yang diperbolehkan untuk masing2 proses
- **Allocation***: n* x *m* matrix. Yg menunjukkan jumlah resource yg **sudah dialokasikan** untk masing2 proses
- **Need***: n* x *m* matrix. Yg menunjukkan sisa resource yg **masih dibutuhkan** oleh tiap2 proses.

Need [*i,j*] = *Max*[*i,j*] – *Allocation* [*i,j*]

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

   Work = *Available*

   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1
2. Find and *i* such that both:

   (a) *Finish* [*i*] = *false*

   (b) $Need_i \leq Work$

   If no such *i* exists, go to step 4
3. *Work* = *Work* + *Allocation*

   Finish[*i*] = *true*

   go to step 2
4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state

Ex of Banker's Alg

- 5 processes P_0 through P_4;
  3 resource types:
  A (10 instances), B (5instances), and C (7 instances)

Snapshot at time 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	

Ex of Banker's Alg (2)

- The content of the matrix *Need* is defined to be *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

- The system is in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfies safety criteria

Cara Perhitungan

- P1 (1,2,2) < (3,3,2) => (3,3,2) + (2,0,0) = (5,3,2)
- P3 (0,1,1) < (5,3,2) => (5,3,2) + (2,1,1) = (7,4,3)
- P4 (4,3,1) < (7,4,3) => (7,4,3) + (0,0,2) = (7,4,5)
- P2 (6,0,0) < (7,4,5) => (7,4,5) + (3,0,2) = (10,4,7)
- P0 (7,4,3) <(10,4,7) => (10,4,7) + (0,1,0) = (10,5,7)

- **Safe!**

- Jika P1 meminta 1 lagi resource A dan 2 resource C, maka request_1 = (1,0,2)

Resource-Request Algorithm for Process P_i

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. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Max_i - Allocation_i;$$

- *If safe* $\Rightarrow$ *the resources are allocated to Pi*
- *If unsafe* $\Rightarrow$ *Pi must wait, and the old resource-allocation state is restored*

Cara Perhitungan

- P1(1,0,2) < P1(1,2,2) = OK
- P1(1,0,2) < Avail(3,3,2) = OK
- Avail = (3,3,2) – (1,0,2) = (2,3,0)
- Allocation P1 = (2,0,0) + (1,0,2) = (3,0,2)
- Need P1 = (3,2,2) – (3,0,2) = (0,2,0)

- Kemudian periksa <P1,P3,P4,P2,P0> = **Safe**

Kelemahan Algoritma Banker

- Proses2 **blm tahu** berapa jml maks resource yg dibutuhkan
- Jml proses **tidak tetap** setiap saat
- Beberapa resource dapat diambil dr sistem sewaktu2, walau secara teori ada, namun kenyataannya tidak ada
  - Contoh : portable storage (flashdisk)
- Algoritma menginginkan untuk memberikan semua permintaan sehingga waktu tak terbatas

Mengatasi Deadlock (Deadlock Prevention)

- **Mutual Exclusion:** yaitu dengan cara tidak berbagi data dengan proses lain atau dengan kata lain menyediakan data sendiri
- **Hold and Wait**: penggunaan resource diperbolehkan jika semua sumber daya yang diperlukan tidak digunakan oleh proses lain
  - Sblm suatu proses memakai resource, dia hrs melepaskan resource yg dibawanya
- **No Preemptive:** jika suatu proses meminta resource, dan blm bs dipenuhi, maka proses tersebut hrs membebaskan resourcenya dulu
- **Circular Wait:** Setiap kebutuhan total didata terlebih dahulu
  - Memberi nomor urut pada tiap resource, dan tiap proses hanya boleh menggunakan resource secara berurutan

Membiarkan deadlock dan mendeteksinya

- When, and how often, to invoke depends on:
  - How **often** a deadlock is likely to occur?
  - How many processes will need to be **rolled back**?

Recovery from Deadlock

- Menggagalkan semua proses yg deadlock
- Membackup dan merestart semua proses yg deadlock
- Menggagalkan semua proses yg deadlock secara berurutan sampai tidak ada yg deadlock
- Mengalokasikan resource secara berurutan sampai tidak ada deadlock

Recovery from Deadlock: Process Termination

- Kriteria mematikan program:
  - Yang memiliki waktu telah berjalan terkecil
  - Yang jumlah outputnya sedikit
  - Yang memiliki sisa waktu eksekusi tersbesar
  - Yang menggunakan total sumber daya terkecil
  - Yang memiliki prioritas terkecil
  - Yang merupakan proses batch
- Setelah dimatikan, resource perlu dikembalikan!
- Data pada program yg dimatikan jg harus **dirollback**!

Mengabaikan Deadlock

- Mengabaikan adanya deadlock dan menganggap keadaan deadlock tidak pernah terjadi (Algoritma **Ostrich**)
  - Secara sederhana algoritma ini dapat dikatakan abaikan deadlock seakan-akan tidak ada masalah apapun dengannya
  - Algoritma ini disadur oleh Sistem Operasi **Unix**, meskipun memerlukan biaya yang cukup besar untuk mengatasi sebuah deadlock

NEXT

- Main Memory Management