# Penggunaan Graf pada Pendeteksian Kasus *Concurrency* yaitu *Deadlock* dan Algoritma Penanganannya

Grace Claudia - 13520078
*Program Studi Teknik Informatika*
*Sekolah Teknik Elektro dan Informatika*
*Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia*
*[1]13520078@mahasiswa.itb.ac.id*

***Abstract*— Saat sebuah sistem operasi berjalan, ada banyak proses yang terjadi. Proses-proses ini disebut *concurrent* apabila diproses dalam saat yang sama. Proses yang mengalami kekongruenan ini dapat terjadi secara individual maupun saling bekerja sama dengan proses lain sehingga membutuhkan koordinasi yang baik antar proses yang berjalan dalam komputer. Salah satu permasalahan kekongruenan pada sistem operasi adalah *deadlock*. Untuk menghindari masalah tersebut, permasalahan *deadlock* ini dapat dideteksi dengan memanfaatkan teori graf.**

***Keywords*— concurrency, deadlock, graf, proses**

## I. PENDAHULUAN

Di dunia yang serba modern sekarang, Sebagian besar penduduk perkotaan di dunia memiliki akses penggunaan komputer yang mereka pakai untuk menunjang kehidupan mereka. Untuk keberjalanan suatu komputer, hal yang terpenting yang harus ada adalah sebuah *operating system* atau sistem operasi. Sistem operasi ini mengatur memori dan melakukan semua proses komunikasi antar perangkat keras (*hardware*) atau perangkat lunak (*software*). Tanpa adanya sistem operasi ini, komputer tidak dapat menjalankan fungsinya dengan baik. Tidak hanya ada di komputer, sistem operasi juga ada di telepon gengam yang kita gunakan.

Gambar 1. Contoh-contoh Operating sistem yang ada (Sumber: https://p4tkpertanian.kemdikbud.go.id/ diakses pada 3 Desember 2021)

Operating sistem memiliki banyak kegunaan diantaranya:

1. Mengeksekusi dan menjalankan suatu program.
2. Mengatur media penyimpanan seperti mengelola seluruh drive yang terpasang pada komputer
3. Mengatur proses berjalannya suatu komputer degan membagikan informasi dan menyinkronisasi proses dalam komputer
4. Menjaga keamanan data dari serangan siber
5. Menampilkanantar muka yang mempermudah pengguna untuk memasukkan dan menerima informasi.

Dalam perancangan sistem operasi sendiri, ada hal yang menjadi landasan umum yaitu *concurrency.* Dalam proses yang terdapat dalam sebuah sistem operasi, apabila berada pada saat yang sama, maka akan disebut *concurrent.* Untuk penanganan kekongruenan ini, diperlukan koordinasi dan sinkronisasi yang baik sehingga antar proses dapat saling berinteraksi dengan baik.

Dalam implementasinya, kekongruenan memiliki beberapa masalah yang harus di selesaikan tak terkecuali masalah *deadlock*. *Deadlock* merupakan keadaan dimana sejumlah proses melakukan permintaan yang tidak bisa dijalankan oleh scheduler karena permintaan-permintaan yang ada saling tunggu menunggu atau bergantung pada proses lain yang belum selesai. *Deadlock* ini merupakan salah satu masalah yang harus ditangani dengan serius dalam *multitasking concurrent programming sistem* agar suatu sistem operasi dapat berjalan bagaimana semestinya. Ada berbagai cara untuk menangani *deadlock*, masing-masing memiliki kelebihan dan kekurangannya sendiri.

Dalam makalah ini, akan dibahasa mengenai bagaimana graf dapat mendeteksi adanya permasalahan *concurrency* yaitu *deadlock* dan algoritma-algoritma lainnya untuk menangani kasus ini. Makalah ini dibuat oleh penulis dengan harapan agar membantu meningkatkan pemahaman pembaca terkait kasus yang sering terjadi pada operating sistem yaitu *deadlock* dan cara menanganinya.

## II. DASAR TEORI

### A. Graf

Graf adalah graf adalah himpunan dari objek-objek yang dinamakan titik, simpul, atau sudut dihubungkan oleh penghubung yang dinamakan garis atau sisi.

Secara matematis, definisi graf adalah sebagai berikut: Graf G = (V, E), yang dalam hal ini:

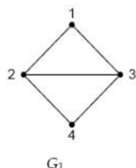
V = himpunan tidak-kosong dari simpul-simpul (vertices) = {v1, v2, ..., vn}

E = himpunan sisi (edges) yang menghubungkan sepasang simpul = {e1, e2, …, en}

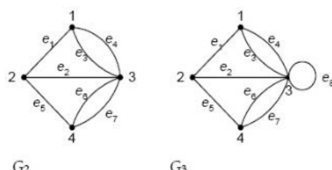
Jenis-jenis graf dibedakan menjadi dua, yaitu berdasarkan pada ada tidaknya gelang atau sisi ganda pada suatu graf dan orientasi arah pada sisi graf.

Bedasarkan gelang atau sisi ganda, graf terbagi menjadi 2 jenis:

1. Graf sederhana
   Graf ini tidak mengandung gelang maupun sisi ganda
2. Graf tak-sederhana
   Graf ini mengandung gelang atau sisi ganda. Graf ini dibagi lagi menjadi dua yaitu:
   1) Graf ganda
      Graf ini mengandung sisi ganda
   2) Graf semu
      Graf ini mengandung sisi gelang



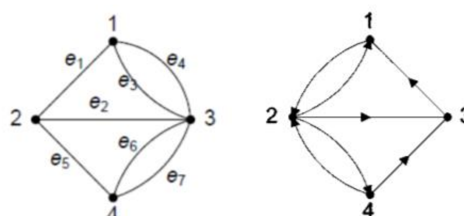

Gambar 2. Contoh graf sederhana
(Sumber: http://athayaniimtinan.blogspot.com/ diakses pada 3 Desember 2021)




Gambar 3. Contoh graf tidak sederhana
(Sumber: http://athayaniimtinan.blogspot.com/ diakses pada 3 Desember 2021)

Bedasarkan orientasi, graf terbagi juga menjadi 2 jenis:

1. Graf tak-berarah
   Graf ini sisinya tidak mempunyai orientasi arah
2. Graf berarah
   Graf ini setiap sisinya diberikan orientasi arah



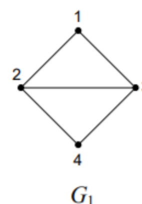

Gambar 4. Jenis graf bedasarkan arah
(sumber: http://denadaapriliaputri.blogspot.com/ diakses pada 3 Desember 2021)

| Jenis | Sisi | Sisi ganda | Sisi gelang |
|---|---|---|---|
| Graf sederhana | Tak berarah | Tidak | Tidak |
| Graf ganda | Tak berarah | Ya | Tidak |
| Graf semu | Tak berarah | Ya | Ya |
| Graf berarah | Berarah | Tidak | Ya |
| Graf granda berarah | Berarah | Ya | Ya |

Tabel 1. Jenis graf dan keterangannya
(sumber: https://informatika.stei.itb.ac.id/ diakses pada 3 Desember 2021)

Terminologi Graf

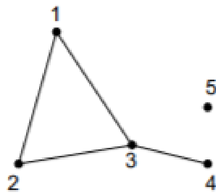
- Ketetanggan (Adjacent)
  Dua buah simpul dikatakan bertetangga bila keduanya terhubung langsung.




Gambar 5. Graph G1
(Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir diakses pada 3 Desember 2021)

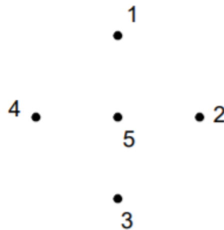
  Pada graf G1, simpul 1 bertetangga dengan simpul 2 dan 3, namun simpul 1 tidak bertetangga dengan simpul 4.

- Bersisian (Incidency)
  Untuk sembarang sisi e = (vj, vk) dikatakan e bersisian dengan simpul vj, atau e bersisian dengan simpul vk
  Pada gambar 5, sisi (2, 4) bersisian dengan simpul 2 dan simpul 4, dan sisi (2, 3) bersisian dengan simpul 2 simpul 3, tetapi sisi (1, 2) tidak bersisian dengan simpul 4
- Simpul terpencil (Isolated Vertex)
  Simpul terpencil ialah simpul yang tidak mempunyai sisi yang bersisian dengannya



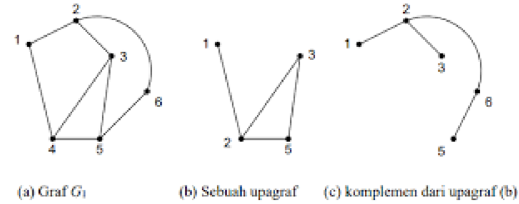

Gambar 6. Graf G3
(Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir diakses pada 3 Desember 2021)
Pada G3, isolated vertex adalah 5.

- Graf kosong (Null Graph)
  Graf yang himpunan sisinya merupakan himpunan kosong



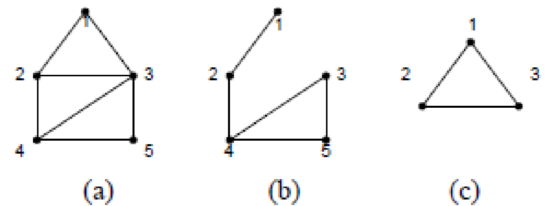

Gambar 7. Graf Gx
(Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir diakses pada 3 Desember 2021)
Gx merupakan graf null.

- Derajat (Degree)
  Derajat suatu simpul adalah jumlah sisi yang bersisian dengan simpul tersebut. Notasi: d(v)
  Pada gambar 5, jumlah derajat pada simpul 2 sama dengan simpul 3 yaitu yaitu d(2) = d(3) = 3, jumlah derajat pada simpul 1 sama dengan simpul 4, yaitu d(1) = d(4) = 2
- Lintasan (Path)
  Lintasan yang panjangnya n dari simpul awal v0 ke simpul tujuan vn di dalam graf G1 pada gambar 5 berselang-seling simpul-simpul dan sisi-sisi yang berbentuk v0, e1, v1, e2, v2, …, vn –1, en, vn sedemikian sehingga e1 = (v0, v1), e2 = (v1, v2), …, en = (vn-1, vn) adalah sisi-sisi dari graf G.
  Pada gambar 5, lintasan 1, 2, 4, 3 adalah lintasan dengan barisan sisi (1,2), (2,4), (4,3). Panjang lintasan adalah jumlah sisi dalam lintasan tersebut. Lintasan 1, 2, 4, 3 pada G1 memiliki panjang 3.
- Sirkuit (Circuit)
  Lintasan yang berawal dan berakhir pada simpul yang sama disebut sirkuit atau siklus. Panjang sirkuit adalah jumlah sisi dalam sirkuit tersebut
  Pada Gambar 5, titik: 1, 2, 3, 1 adalah sebuah sirkuit.
- Terhubung (Connected)
  Dua buah simpul v1 dan simpul v2 disebut terhubung jika terdapat lintasan dari v1 ke v2. G disebut graf terhubung (connected graph) jika untuk setiap pasang simpul vi dan vj dalam himpunan V terdapat lintasan dari vi ke vj. Jika tidak, maka G disebut graf tak-terhubung (disconnected graph).
- Upagraf (subgraph) dan komplemen Upgraf
  Graf G1 = (V1, E1) dan graf G2 = (V2, E2) dikatakan upagraf jika V2 merupakan himpunan bagian dari V1 dan E2 merupakan himpunan bagian dari E1.
  Komplemennya adalah suatu graf G3 dimana E3 = E1 – E2 dan V3 adalah himpunan simpul yang anggota-anggota E3 bersisian dengannya.



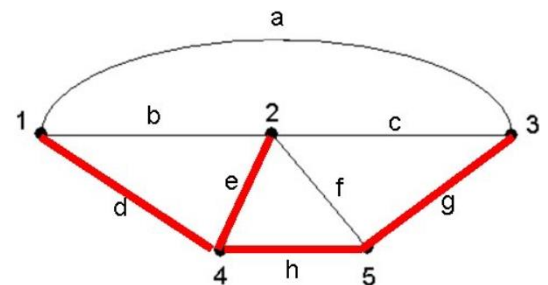

Gambar 8. Upagraf dan komplemennya
(Sumber: https://informatika.stei.itb.ac.id/ diakses pada 3 Desember 2021)

- Upagraf merentang
  Upagraf G1 = (V1, E1) dari G2 = (V2, E2) dikatakan upagraf rentang jika V1 = V2 (yaitu G1 mengandung semua simpul dari G2).



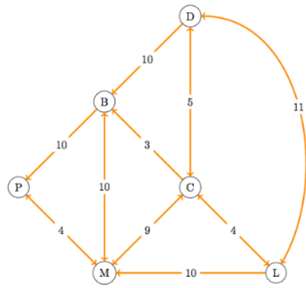

Gambar 9. Upagraf merentang
(Sumber: https://ayobelajarbos.blogspot.com/ diakses pada 3 Desember 2021)

- Cut-set
  Cut-set dari graf terhubung yaitu G merupakan himpunan sisi yang jika dibuang dari graf G akan menyebabkan graf G menjadi tidak terhubung




Gambar 10. Cutset sebuah graf (garis merah)
(Sumber: https://slidetodoc.com diakses pada 3 Desember 2021)

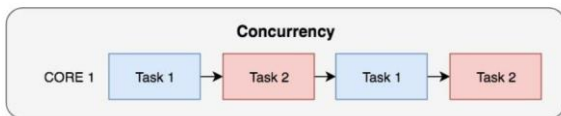
- Graf berbobot
  Graf yang setiap sisinya memiliki harga/bobot.



Gambar 11. Graf berbobot
(Sumber: https://newbedev.com diakses pada 3 Desember 2021)

B. *Concurrency*

*Concurrency* adalah kemampuan suatu program untuk memproses *multiple order* atau *request.* Analoginya seperti 2 antrian pada 1 kasir. *Concurrency* ini dapat meningkatkan kecepatan eksekusi secara keseluruhan dalam sistem yang multi prosessor dan multi core.



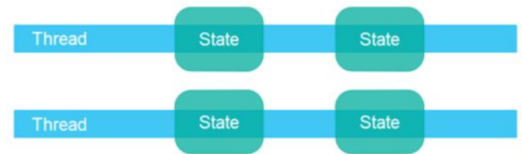
Gambar 12. Diagram penggambaran *concurrency*
(Sumber: https://dev.to diakses pada 3 Desember 2021)

Contoh dari *concurrency* adalah:
- Beberapa computer pada suatu jaringan yang sama
- *Multiple processor* pada satu komputer
- Beberapa aplikasi yang berjalan pada satu computer
- Website yang harus menghandle *request* dari banyak user

Model pada *Concurrency*:
- Shared state: thread dalam sistem berbagi beberapa state di antara mereka, state ini adalah kumpulan dari data – data

Gambar 13. Diagram penggambaran shared state
(Sumber: https://dev.to diakses pada 3 Desember 2021)

- Separate state: thread yang berbeda tidak berbagi state apapun. Jika dilakukan komunikasi antar state yang berbeda, mereka menukar objek yang immutable (tidak dapat diubah)/mengirim Salinan data.



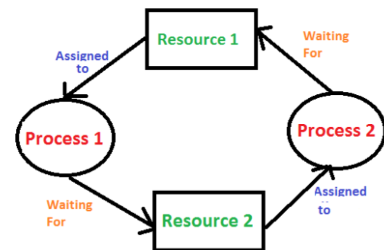
Gambar 14. Diagram penggambaran separate state
(Sumber: https://dev.to diakses pada 3 Desember 2021)

C. *Deadlock*

*Deadlock* merupakan keadaan yang menggambarkan dua proses atau lebih yang saling menunggu proses yang lain untuk mendapatkan *resource* yang dibutuhkan. Karena proses-proses itu saling menunggu, maka prosess tersebut mengalami gangguan yang berupa berhentinya proses atau tidak majunya pekerjaan dari suatu proses. *Deadlock* ini dapat diartikan juga kebuntuan proses dari suatu sistem operasi Ketika ada beberapa proses yang me-*request* sebuah *resource* yang hanya boleh dirubah oleh satu proses dalam satu waktu.

Sebuah proses dalam operating sistem menggunakan *resource* sebagai berikut:
1. *request* sebuah *resource*
2. memakai *resource* yang telah di-*request*
3. melepas *resource*

Jika terjadi *deadlock*, maka proses-proses ini akan terganggu.



Gambar 15. Diagram contoh *deadlock*
(Sumber: https://www.geeksforgeeks.org/ diakses pada 4 Desember 2021)

Pada gambar 15, *deadlock* terjadi karena proses 1 memegang *resource* 1 dan menunggu *resource* 2 yang dibutuhkan oleh proses 2, padahal proses 2 menunggu proses 1.

Ada 2 tipe *deadlock*:
- *AND condition*: sebuah proses yang membutuhkan proses lain untuk eksekusi yang dapat dieksekusi jika sebah proses memiliki semua sumber data yang dibuthkan
- *OR condition*: sebuah proses yang membutuhkan sumber data untuk dieksekusi yang dapat dijalankan jika minimal ada terdapat satu data yang dibutuhkand dari sumber data

Cara penanganan *deadlock* ada 4:

- *deadlock avoidance:* tidak membiarkan suatu proses mamsuk ke *deadlock* state
- *deadlock prevention:* sama dengan *deadlock* avoidance pada dasarnya tetapi beda cara penanganannya.
- *deadlock detection and recovery:* membiarkan *deadlock* terjadi lalu melakukan *preemption* untuk menghandle setelah terjadi
- *ignore:* jika suatu keadaan *deadlock* terjadi sangat jarang, maka akan dibiarkan terjadi dan hal yang kita lakukan adalah mereboot sistem.

Penanganan akan lebih lanjut dibahas pada bab III.

## III. PENANGANAN *DEADLOCK*

### A. *Deadlock Avoidance*

Dalam penanganan ini, sebuah sumber diperbolehkan berjalan saat proses tersebut menghasilkan *state* yang aman. Setiap bagian dari proses yang berjalan harus memantain *global state.* Pengecekan untuk *state* yang aman harus dengan *mutual exclusion.* Dalam penanganan sehari, hari *deadlock avoidance* bukan solusi praktikal. *Deadlock avoidance* ini dapat dilakukan salah satunya dengan *banker algorithm.*

Banker Algorithm

Algoritma Banker dikemukakan oleh Edsger W. Djikstra yang merupakan salah satu cara untuk menghindari *deadlock*. Algoritma ini disebut algoritma banker karena dimodelkan, ada sebuah bank di kota kecil yang berurusan dengan sekupulan nasaban yang memohon kredit. Analoginya, nasabah merupakan proses-proses yang berjalan sedangkan uang merupakan sumbe daya, bankir merupakan sistem operasi.

Setiap nasabah memiliki batas dalam mengajukan kredit. Apabila seorang nasabah telah sampai pada batas untuk melakukan peminjaman kredit, maka diasumsikan bahwa orang itu sudah mengakhiri masalah-masalah bisnisnya dan dengan segera dapat mengembalikan pinjaman pada bank. Tiap-tiap nasabah tersebut dapat mengajukan peminjaman kredit pada saat apapun. Bankir dapat menolak atau menyetujui permohonan kredit ini sesuai dengan regulasi yang ada. Jika ditolak, maka nasabah yang masih memiliki dana yang telah dipinjamkan untuk mereka dan menunggu selama waktu hingga sampai permohonannya dapat disetujui. Bankir hanya memberikan permintaan yang menghasilkan state yang aman sesuai dengan regulasi. Permohonan kredit yang diajukan ini akan menghasilakan state yang tidak aman sampai permohonan tersebut dipenuhi.

Input yang dibutuhkan algoritma banker:

1. Kebutuhan maksimal *resource* dari setiap proses
2. Alokasi *resource* oleh setiap proses
3. Free *resource* maksimal yang tersedia pada sistem

Struktur data untuk menggunakan banker algorithm:

1. *Available*
   *Available* digambarkan dengan struktur data array dengan Panjang n. Array ini merepresentasikan banyaknya jumlah *resource* yang tersedia pada setiap tipe. Jika Available[i] = x, maka x ada instance yang tersedia dari sebuah *resource* bertipe Ri.
2. *Max*
   Max merupakan matriks berukuran n x m yang merepresentasikan jumlah maksimum sebuah instance dari sebuah *resource* yang dapat me*request* sebuah proses. Jika Max[i][j] = x, maka proses Pi dapat me*request* x instance dari *resource* type Rj.
3. *Allocation*
   Allocaition digambarkan dengan matriks berukuran n x m yang merepresetansikan banyaknya *resource* dari setiap tipe yang sedang dialokasikan terhadap setiap proses. Jika Alokasi[i][j] = k, maka prses Pi yang sedang teralokasi adalah K instance dari *resource* bertipe Rj.
4. *Need*
   Need digambarkan dengan array 2 dimensi yang berukuran n x m. Array of array ini mengindikasi *resource* yang tersisa dari setiap rposes. Jika need[i][j] = k, maka setiap proses Pi akan membutuhkan k lagi instance bertipe Rj untuk menyelesaikan suatu proses yang sedang berjalan.
   Need[i][j] = Max[i][j] - Allocation[i][j]

*Request* hanya dijalankan dengan kondisi sebagai berikut:

1. Jika *request* yang dibuat oleh sebuah proses kurang dari atau sama dengan jumlah maksimalyang diperlukan oleh proses tersebut
2. Jika *request* yang dibuat sebuah proseskurang dari sama dengan yang *resource* yang tersedia pada sistem.

Persetujuan atau penolakan permohonan kredit ditentukan dengan menggunakan algoritma safety dan algoritma *resource request*.

- *Safety algorithm*
- *Resource request algorithm*

| Total instances of each resources | | |
|---|---|---|
| R1 | R2 | R3 |
| 5 | 5 | 5 |

| Process | Allocation | | | Max | | |
|---|---|---|---|---|---|---|
| | R1 | R2 | R3 | R1 | R2 | R3 |
| A | 1 | 2 | 1 | 2 | 2 | 4 |
| B | 2 | 0 | 1 | 2 | 1 | 3 |
| C | 2 | 2 | 1 | 3 | 4 | 1 |
| Total_alloc | 5 | 4 | 3 | | | |

Available = Total - Total_alloc
= [5, 5, 5] - [5, 4, 3]
= [0, 1, 2]

Gambar 16. Gambaran singkat algoritma banker
(Sumber: https://www.afteracademy.com/ diakses pada 12 Desember 2021)

Kekurangan dari algoritma ini diantaranya:

- Sejalan dengan masuknya proses ke sistem, ia akan memprediksi banyaknya maksimum *resource* yang dibutuhkan sehingga sangat tidak praktikal,
- Di algoritma ini, banyaknya proses akan sama seiring berjalannya waktu yang tidak memungkinkan adanya sistem yang interaktif

- Algoritma ini memubutuhkan adanya angka yang pasti dari sebuah *resource* untuk dialokasikan. Jika sebuah perangkat rusak dan tidak-tibak tidka tersedia maka algoritma ini tidak akan berjalan
- Eksekusi program sangatlah mahal saat ada banyak *resource* dan proses.

*B. Deadlock prevention*

Dalam *deadlock preention,* semua *resource* ada pada saat yang sama. Teknik ini mengatasi sebuah proses dari pegnambilan data selama menunggu data dari sumber. Hal ini dianggap tidak efisien karena bisa terjadi saat fase membutuhkan sumber data, fase ini tidak dapat diprediksi sehingga tidak efisien dan bukan solusi universal.

Karakteristik dari *deadlock* diantaranya adalah:

1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait

Ide dari penanganan inni adalah mengeliminasi keempat karakteristik dari *deadlock* itu sendiri.

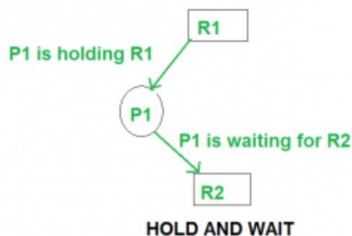
Pengeliminasian *Mutual Exclusion*

Sebenarnya tidak mungkin untuk meng-cancel terjadinya mutual exclusion karena beberapa *resource* seperti tape drive dan printer sebenarnya tidak dapat bekerja dengan membagi sesuatu.

Pengeliminasian *hold and wait*

Cara pengeliminasian *hold and wait* adalah:

1. Mengalokasi semua *resource* yang dibutuhkan ke proses sebelum mulainya eksekusi, dengan cara ini, kondisi hold and wait dapat dieliminasi tapi akan menyebabkan rendahnya utilisasi perangkat. Misalnya, jika sebuah proses membutuhkan printer dan kita telah mengalokasika printer sebelum memulai eksekusi, printer tersebu akan selamanya berhentii sampai printer tersebut sudah melakukan eksekusi
2. Proses ini akan membuat *request* baruu untuk *resource* setelah melepas set dari *resource* pada saat ini.




Gambar 17. Gambaran singkat hold and wait
(Sumber: https://www.geeksforgeeks.org/ diakses pada 13 Desember 2021)

Pengeliminasian Circular Wait

Setiap *resource* akan diberikan nomor. Sebuah proses dapat meminta *resource* sesuai urutan membesr ato mengecil dari sebuah nomor. Misalnya jika proses P1 di alokasi di *Resource* R5, nantinya jika P1 meminta R4, R3 kurang dari R5 sehingga *request* tersebut tidak akan berjalan, hanya *request* yang lebih dari R5 yang berjalan.

Pengeliminasian No preemption

*Resource* yang didahului dari sebuah proses jika sebuah *resource* dibutuhkan oleh proses dengan prioritas tinggi lainnya.

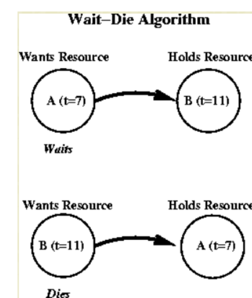
Ada dua algoritma berbasis metode *deadlock prevention* dengan berbasis pengeliminasian kondisi no preemption :

1. *Wait-Die Algorithm*
2. *Wound-wait Algorithm*

Kedua algoritma ini dikenalkan oleh Rosenkrantz(1978).

*Wait-Die Algorithm*

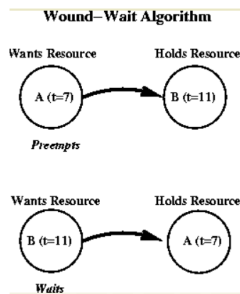
- Hanya mengizinkan menunggu jika proses yang menunggu lebih dulu
- Karena timestamp bertambah pada setiap ranai proses, maka cycle tidak diperbolehkan
- Algoritma ini mengcancel proses yang lebih terdahulu
- Saat proses yang terdahulu berjalan kembali, dan meminta *resource* lagi, akan dicancel kembali
- Lebih tidak efisien disbanding wound-wait algorithm




Gambar 18. Gambaran wait-die algorithm
(Sumber: https://www.cs.colostate.edu/ diakses pada 13 Desember 2021)

*Wound wait algorithm*

- Sebaliknya, algoritma ini menunggu proses yang lebih dahulu
- Disini, *timestamps* berkurang disetiap rantai proses yang menunggu, jadi cycle tidak mungkin terjadi lagi
- Di algoritma ini, kita akan memberi proses yang belakangan prioritas
- Pada algoritma ini, proses yang lebih dahulu mendahului proses yang lebih lama.
- Saat proses yang lebih dahulu meminta ulang *resource*, maka harus menunggu proses yang selanjutnya untuk selesai




Gambar 19. Gambaran wound-wait algorithm
(Sumber: https://www.cs.colostate.edu/ diakses pada 13 Desember 2021)

*C. Deadlock detection and recovery*

Struktut data yang digunakan:

- *Available*
  Struktur data yang menggambarkannya merupakan sebuah vector dengan Panjang m yang mengindikasi banyaknya *resource* yang tersedia dari setiap tipe
- *Allocation*
  Merupakan matrix n x m yang menggambarkan jumlah *resource* dari tiap tipe yang sedang dialokasikan ke setiap proses
- *Request*
  Matriks n x m yang mengindikasi *request* yang ada dari setiap proses. Jika *request*[i][j] = k, maka proses Pi meminta k lagi instance bertipe Rj

Algoritma *deadlock detection*:

1. Menginisiasi *work* dan *finish* sebagai vector dengan Panjang m dan n lalu menginsisiasi *work* sebagai *availlale dan* untuk I dari 1 sampai, jika alokasi !=0 maka finish[i] = false, selain itu finish[i] = true.
2. Mencari index i yang memenuhi finish[i]= false atau *request*[i] <= work, jika tidak ada maka dilanjut ke step 4
3. Work = work + allocation[i], finish[i] = true, ke step kedua
4. Jika finish[i] = false untuk sejumlah i dimana i 1 <= i <= nm maka sistem dalam keadaan *deadlock*. Sebagai tambahan, jika finish[i] = faslem maka Pi *deadlock*

Proses Recovery (Terminasi):

- Mengagalkan semua proses *deadlock* atau mengagalkan satu proses *deadlock* hingga setiap siklus *deadlock* tereliminasi
- Kondisi urutan pengagalan:
  - Prioritas proses
  - Umur proses dan waktu yang tersisa
  - *Resource* yang telah dipakai oleh sebuah proses
  - *Resource* dari proses yang perlu diselesaikan
  - Rouses yang telah digunakan
  - Berapa banyak proses yang harus diterminasi

*D. Ignore*

Di banyak sistem, *deadlock* terjadi cukup jarang. Dengan alasan biaya dan rendahnya frekuensi kejadian *deadlock*, banyak yang beranggpaan untuk apa mengaplikasikan berbagai metode untuk penanganan *deadlock*. Kita harus mengorbankan salah satu diantara kebenaran dan performa computer. Jika kita mau tingkat kebenaran yang cukup tinggi pada sistem kita, maka CPU harus akan mengorbankan kecepatan performa computer untuk mengecek *deadlock*. Tapi, jika kita tidak memedulikan *deadlock*, kita dapat dengan mudah merestart sistem kita saat terjadinya *deadlock*. Kelemahannya adalah kita dapat menghapus data yang belum kita simpan. Pemilihan untuk tidak memedulikan *deadlock* atau melakukan penanganan pada masalah ini snagta tergantung pada situasi yang dibutuhkan. Jika sistem menangani data yang sangat amat penting maka, disarankan menggunakan *deadlock* handling.

## IV. *DEADLOCK* DETECTION DENGAN GRAF

Pendeteksian *deadlock* dapat dilihat dari juga dari graph yang menggambarkan proses atau *request*nya. Graph yang dimaksud ada dua yaitu *resource-allocation graph (RAG)* dan *wait-for-graph (WFG)*. Perbedaan yang signifikan dari kedua graph tersebut adalah pada WFG, tidak digambarkan *resource*nya sehingga hanya merepresentasikan prosesnya saja. Sebaliknya, RAG merepresentasikan baik proses maupun *resource*nya.

*Resource-Allocation Graph (RAG)*

RAG ini menjelaskan apa state dari sistem di dalam context proses dan *resource*. Berapa banyak *resource* yang available, berapa banyak yang dialokasikan, dan apa *request* dari setiap proses. Kelebihan penggunaan graph adalah kita dapat langsung dengan mudah melihat apakah terjadi *deadlock* atau tidak lewat graph. Graph ini lebih menguntungkan untuk sistem dengan proses dan *resource* yang sedikit. Jika sudah cukup banyak maka dianjurkan menggunakan tabel.

Simpul dan sisi RAG

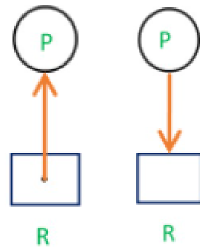
Di dalam RAG ada 2 tipe simpul:

1. Simpul proses: setiap proses akan direpresentasikan sebagai satu simpul. Secara general, setiap proses akan direpresentasikan dengan sebuah lingkaran
2. Simpul *resource*: Setiap *resource* akan direpresentasikan sebagai satu simpul *resource*. simpul *resource* dibagi lagi menjadi 2 jenis:
   1) single instance *resource*: akan direpresentasikan sebagai sebuah box, didalam box tersebut akan ada satu titik. Jadi, banyaknya titik akan mengindikasi berapa instance yang ada di setiap tipe *resource*, contoh: CPU
   2) multi-*resource* instance: akan direpresentaiskan sebagai box juga, di dalam box akan ada banyak dots yang direpresentasikan, contoh: Register

Gambar 20. Gambaran single (kiri) dan multi-*resource* instance (kanan)
(Sumber: https://www.geeksforgeeks.org/ diakses pada 13 Desember 2021)

RAG memiliki dua jenis sisi berarah:

1. Sisi assign: Jika sudut mengassign sebuah *resource* ke suatu proses maka akan disebut sisi asign
2. Sisi *request*: ini menggambarkan akan ada esbuah *resource* yang ingin kita selesaikan esekusinya

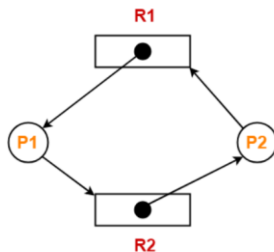



Gambar 21. Gambaran sisi assign (kiri) dan sisi *request* (kanan)
(Sumber: https://www.geeksforgeeks.org/ diakses pada 13 Desember 2021)

Dapat disimpulkan, jika sebuah proses menggunakan *resource* maka akan ada panah yang digambarkan dari simpul nreouse ke simpul proses. Sebaliknya, jika sebuah proses meminta *resource*, maka sebuah panah akan digambarkan dari sebuah simpul proses ke simpul *resource*.

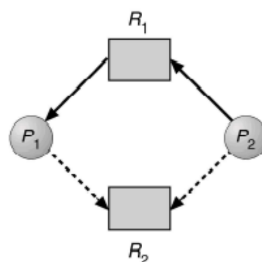
Cara mengidentifikasi *deadlock*

Dalam kasus dimana semua *resource* memiliki single isntance, jika terbentuk cycle maka **sudah pasti** sistem tersebut terjadi *deadlock*




Gambar 22. Gambaran terjadinya *deadlock* pada sistem
(Sumber: https://www.gatevidyalay.com/ diakses pada 13 Desember 2021)

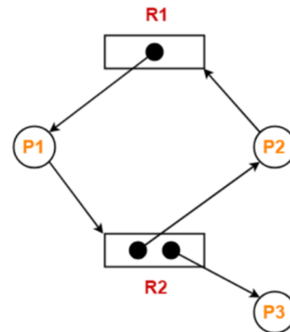
Digambarkan pada gambar 21, proses 1 me*request* R2 yang membutuhkan proses P2, proses P2 diassign oleh R1 yang membutuhkan P1 sehingga terjadi proses *deadlock*.Untuk itu pada gambar 23, akan diberikan graf untuk menghindari terjadinya *deadlock*.




Gambar 23. RAG untuk menghindari *deadlock*
(Sumber: http://csit.udc.edu// diakses pada 13 Desember 2021)

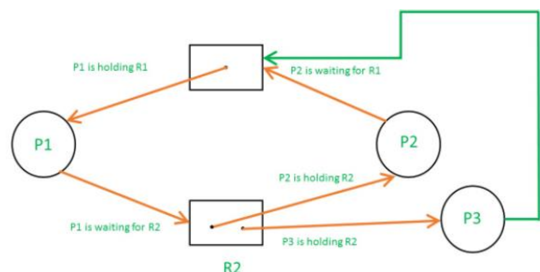
Dalam kasus dimana *resource* merupakan multi-*resource* maka jika ada siklus dalam graph maka sistem ada kemungkinan dalam *deadlock* state. Tapi, tidak dijamin *deadlock*. Pengecekan akan dilakukan dengan banker algorithm dengan bantuan pengisian tabel yaitu allocation and *request* matrix.

Jadi, dapat disimpulkan adanya siklus merupakan **syarat perlu** dari terjadinya *deadlock* di multi-*resource* instance tetapi **bukan syarat pasti.**




Gambar 24. Gambaran sistem tidak *deadlock* pada sebuah graph yang memiliki siklus
(Sumber: https://www.gatevidyalay.com/ diakses pada 13 Desember 2021)

Digambarkan pada Gambar 24, P3 tidak membutuhkan *resource* jadi dapat langsung selesai dieksekusi. Setelah eksekusi P3 melepaskan *resource*nya, selanjutnya P1 mengalokasi *resource* yang diminta, setelah selesai maka akan menyelesaikan eksekusi dan melepaskan *resource*, begitu juga dengan P2, Maka sistem ini aman dari *deadlock* karena P3, P2, dan P1 dapat dieksekusi..




Gambar 25. Gambaran graf yang merepresentasikan terjadinya *deadlock* pada sistem
(Sumber: https://www.geeksforgeeks.com/ diakses pada 13 Desember 2021)

Dengan menggunakan matriks alokasi dan matrix *request* sesuai dengan yang telah dijelaskan pada bab sebelumnya maka tabel dari graf pada gambar 25 akan digambarkan sebagai berikut:

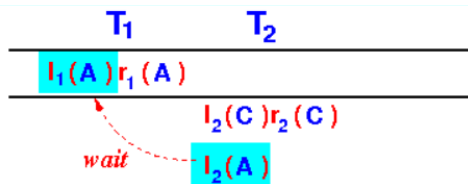
| Proses | Alokasi *Resource* | | *Request* *Resource* | |
|---|---|---|---|---|
| | R1 | R2 | R1 | R2 |
| P1 | 1 | 0 | 0 | 1 |
| P2 | 0 | 1 | 1 | 0 |
| P3 | 0 | 1 | 1 | 0 |

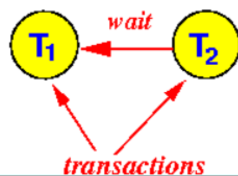
Tabel 2. Table filling dengan banker algorithm

*Resource* yang available adalah (0,0) padahal dibutuhkan (0,1), (1,0), dan (1,0) maka kita tidak dapat memenuhi 1 kreiteria. maka terjadi *deadlock*.

*Wait-For Graph (WFG)*

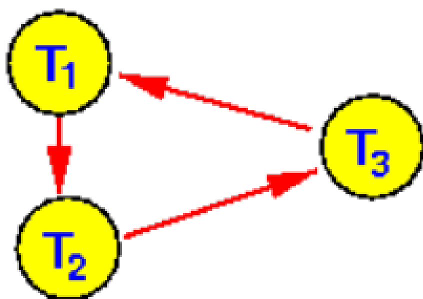
Selanjutnya, deteksi *deadlock* dapat juga dilakukan dengan memanfaatkan wait-for-graph. Simpul dari WFG merepresentasikan transaksi, sedangkan sisi misalnya I → J dari WFG merepresentasikan bahwa transaksi I menunggu *resource* yang dimiliki oleh transaksi J.



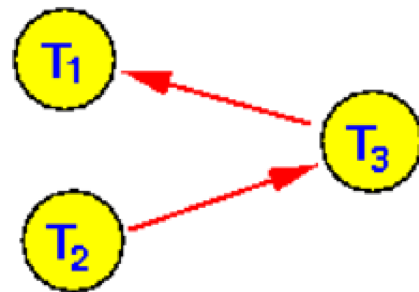

Gambar 26. Contoh gambaran transaksi yang ada dan WFGnya
(Sumber: http://www.mathcs.emory.edu/ diakses pada 14 Desember 2021)

Keadaan *deadlock* pada sebuah sistem ditandai dengan adanya kondisi circular wait atau adanya siklus. Keadaan ini menggambarkan setiap transaksi menunggu transaksi lain untuk selesai yang menyebabkan tidak ada transaksi yang dapat dieksekusi




Gambar 27. Keadaan circular wait
(Sumber: http://www.mathcs.emory.edu/ diakses pada 14 Desember 2021)

Sebaliknya, jika WFG tidak memiliki siklus maka tidak ada *deadlock*.



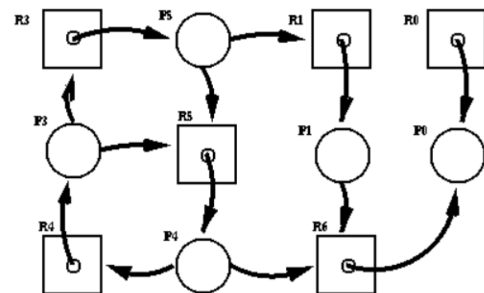

Gambar 28. WFG tanpa *deadlock*
(Sumber: http://www.mathcs.emory.edu/ diakses pada 14 Desember 2021)

Urutan pengeksekusian pada gambar diatas adalah sebagai berikut:

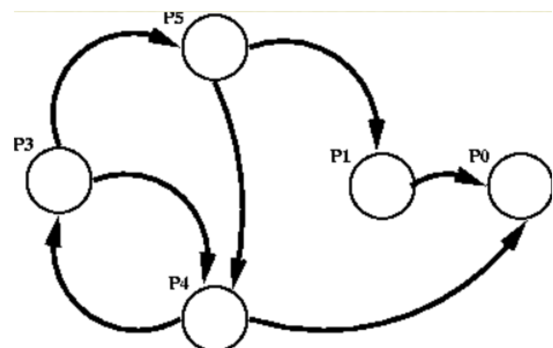
1. Transaksi T1 akan terlebih dahulu dieksekusi dan melepas *resource*nya
2. Selanjutnya, transaksi T3 mendapatkan *resource*nya setelah transaksi T1 selesai dan T3 melepas *resource*nya
3. Di tahap terakhir, transaksi T2 telah memilikii *resource* setelah transaksi T3 selesai.

Mengubah RAG menjadi WFG

Kelebihan dari menggunakan WFG adalah kita dapat dengan mudah melihat adanya siklus dan simpul untuk mendeteksi *deadlock* dalam graf. Kita dapat dengan mengubah RAG menjadi WFG dengan cara menghlangkan node *resource* pada RAG. Contoh dari pengubahannya ada di gambar dibawah ini




Gambar 29. Sebuah single *resource* allocation graph
(Sumber: https://www.cs.colostate.edu/ diakses pada 14 Desember 2021)




Gambar 30. Sebuah WFG dari graph pada gambar 26
(Sumber: https://www.cs.colostate.edu/ diakses pada 14 Desember 2021)

Perlu diingat sebagai catatan penting, pengubahan ini hanya berlaku pada single-instance *resource* RAG.

## V. KESIMPULAN

Teori graf memiliki banyak sekali kegunaan di kehidupan sehari-hari tak terkecuali di dunia komputer. Di setiap komputer terdapat sistem operasi yang mana kadang memiliki masalah. Suatu masalah dimana dua proses atau lebih saling menunggu proses yang lain untuk memberikan *resource* yang dipakai dalam suatu sistem operasi disebut *deadlock*. Salah satu pemanfaatan dari graf dapat digunakan dalam baik pendeteksian *deadlock* dalam suatu sistem operasi. Dalam *single instance resource* adanya siklus dalam graf merupakan syarat pasti terjadinya *deadlock,* sedangkan pada *multi resource instance* adanya siklus dalam graf merupakan syarat perlu terjadinya *deadlock*.

## VI. UCAPAN TERIMA KASIH

Pertama-tama, penulis mengucapkan terimakasih kepada puji syukur kepada Tuhan Yang Maha Esa atas berkat dan rahmatnya, penulis dapat menyelesaikan makalah ini dengan baik dan tepat waktu. Tak lupa juga, penulis mengucapkan terima kasih kepada orang tua yang selalu mendukung penulis dalam segala kegiatan pendidikan yang dijalankan penulis.
Terima kasih sebesar-besarnya juga dipanjatkan penulis kepada Ibu Harlili selaku dosen yang membimbing penulis dalam pembelajaran matematika diskrit selama satu semester ini. Terimakasih juga kepada Ibu Ulfa dan Pak Rinaldi Munir yang sudah bekerjasama dalam keterlaksanaannya perkuliahan IF2120 dengan baik bagi mahasiswa IF Angkatan 2020.
Tak lupa juga, penulis mengucapkan terima kasih kepada teman-teman penulis yang selalu mendukung, memotivasi, dan memberi masukan dalam setiap pembelajaran pada mata kuliah IF2120 ini.

## REFERENSI

[1] https://kumparan.com/how-to-tekno/operating-sistem-definisi-fungsi-dan-jenisnya-1vx7JnWiDbf/1 diakses pada 3 Desember 2021
[2] https://www.geeksforgeeks.org/*deadlock*-in-dbms/ diakses pada 3 Desember 2021
[3] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2021-2022/matdis21-22.htm diakses pada 3 Desember 2021
[4] https://www.gurupendidikan.co.id/*deadlock*-dan-starvation/ diakses pada 4 Desember 2021
[5] https://www.studytonight.com/operating-sistem/bankers-algorithm diakses pada 13 Desember 2021
[6] https://www.cs.colostate.edu/~cs551/CourseNotes/*Deadlock*/DDCMHAlg.html diakses pada 13 Desember 2021
[7] http://csit.udc.edu/~byu/COSC4740-01/Lecture7.pdf diakses pada 13 Desember 2021
[8] https://www.geeksforgeeks.org/*resource*-allocation-graph-rag-in-operating-sistem diakses pada 13 Desember 2021
[9] https://repository.dinus.ac.id/docs/ajar/7-*deadlock*.pdf diakses pada 13 Desember 2021
[10] https://www.cs.colostate.edu/~cs551/CourseNotes/*Deadlock*/WFGs.html diakses pada 14 Desember 2021
[11] http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/8-recv+serial/*deadlock*-waitfor.html diakses pada 14 Desember 2021

## 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, 3 Desember 2020

Grace Claudia 13520078