Penggunaan Brute Force untuk Mendeteksi Potensi Terjadinya Deadlock

Rafi Ramadhan - 13512075
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
rafi.isakh@students.itb.ac.id

Algoritma brute force dikenal sebagai algoritma yang sederhana karena mampu menyelesaikan persoalan secara langsung dengan metode yang mudah dimengerti. Berkaitan dengan karakteristiknya tersebut, hampir semua persoalan dapat diselesaikan dengan algoritma ini. Namun, solusi yang dihasilkan belum tentu optimal.

Seperti yang telah kita ketahui, semua proses yang terjadi dalam komputer diatur oleh suatu mekanisme yang disebut sistem operasi. Sistem operasi mengatur sedemikian rupa proses-proses yang sedang berjalan sehingga tidak terjadi benturan (collision) antarproses. Meskipun begitu, hal-hal semacam itu masih bisa terjadi, salahsatu contohnya adalah deadlock. Biasanya, setiap sistem operasi memiliki algoritma yang berbeda untuk menangani kondisi deadlock.

Melalui karya tulis ini, penulis ingin mencoba menggunakan pendekatan brute force untuk menangani deadlock karena karakteristiknya yang dianggap mampu menyelesaikan hampir semua persoalan. Khusus dalam kasus ini, brute force akan digunakan untuk memeriksa jika suatu kondisi berpotensi untuk menyebabkan deadlock.

Kata kunci: brute force, deadlock, optimal, potensi .

I. PENDAHULUAN

Sistem operasi memiliki mekanisme yang sangat rumit. Dapat dibayangkan bagaimana suatu sistem harus mengatur dan menangani semua proses yang berlangsung dalam komputer. Sehubungan dengan hal tersebut, penulis ingin mengetahui penanganan *deadlock* secara lebih mendalam. Menurut penulis, deadlock merupakan hal yang cukup fatal. Namun, sistem operasi seperti Linux dan Windows menangani deadlock dengan cara mengabaikannya. Maksud dari mengabaikan tersebut adalah sistem berpura-pura "tidak tahu" ketika *deadlock* terjadi dengan asumsi kemungkinan terjadinya *deadlock* sangatlah kecil.

Di sisi lain, penulis cukup tertarik dengan algoritma *brute force*. Algoritma ini adalah satu-satunya algoritma yang sering digunakan untuk menyelesaikan suatu masalah ketika pemrogram sudah tidak bisa lagi memikirkan "solusi pintar" untuk sebuah persoalan. Selain itu, ada beberapa persoalan tertentu yang hanya bisa diselesaikan secara *brute force* meskipun optimasi dari algoritma ini cukup rendah.

Oleh karena itu, penulis ingin menggunakan algoritma *brute force* untuk menangani masalah yang cukup kompleks, yaitu mendeteksi kondisi *deadlock*. Di samping menganalisis kinerja brute force untuk persoalan ini, penulis juga ingin membuktikan karakteristik algoritma *brute force* yang bisa menyelesaikan hampir semua persoalan.

II. ALGORITMA BRUTE FORCE

Algoritma *brute force* merepresentasikan sebuah gaya pemrograman yang primitif [1]. Sebutan ini bukannya tanpa alasan. Dengan menggunakan algoritma ini, pemrogram lebih bergantung kepada kemampuan komputer untuk melakukan komputasi daripada menggunakan kemampuan berpikirnya untuk menyederhanakan persoalan. Memang cara ini tidak salah, namun solusi yang didapat mungkin akan lebih optimal baik dari segi waktu maupun hasil yang didapat jika pemrogram tidak bergantung penuh terhadap kemampuan komputer.

Selain disebut pemrograman primitif, *brute force* juga sering disebut sebagai pemrograman naïf (*naïve programming*) [2]. Hal ini disebabkan metode *brute force* yang seringkali mengabaikan ukuran skala persoalan. Maksudnya, ketika ada suatu metode yang cocok untuk suatu persoalan skala kecil, metode tersebut akan langsung digunakan untuk menyelesaikan persoalan serupa meskipun skalanya jauh lebih besar. Prinsip seperti inilah yang membuat algoritma *brute force* kurang efisien dalam menyelesaikan persoalan. Kode *brute force* dianggap sebagai bentuk penulisan yang membosankan, penuh dengan pengulangan, dan sama sekali tidak memiliki sisi elegan.

Beberapa persoalan yang biasa diselesaikan dengan algoritma brute force, diantaranya pencarian elemen terbesar/terkecil, perkalian dua buah matriks, dan pengurutan elemen. Seperti yang dijelaskan sebelumnya, algoritma *brute force* cukup efektif untuk persoalan skala kecil. Dengan demikian, dapat disimpulkan bahwa algoritma ini akan tidak optimal jika ukuran data yang dimasukkan sangat besar. Terlepas dari itu, algoritma brute force memiliki beberapa kelebihan dan juga kekurangan [2].

Dalam hal kelebihan, *brute force* mempunyai daya cakup penggunaan yang luas (*wide applicability*), yaitu

dapat digunakan untuk memecahkan sebagian besar masalah. Kemudian, algoritma ini mudah dimengerti dan sederhana. Selain itu, metode brute force juga menghasilkan algoritma standar untuk beberapa tugas komputasi seperti pencarian, pengurutan, pencocokan *string*, dan penjumlahan/perkalian n buah bilangan.

Di samping ketidakmampuannya untuk menangani persoalan berskala besar, masih ada lagi beberapa kekurangan dari algoritma ini. Algoritma *brute force* tidak diterima untuk beberapa persoalan karena kinerjanya yang lambat. Kekurangan lainnya adalah alur prosesnya yang dianggap kurang konstruktif atau kurang kreatif jika dibandingkan dengan trknik pemecahan masalah lainnya.

Jika dilihat secara sepintas, algoritma brute force tidak layak untuk dijadikan salah satu teknik pemecahan masalah. Namun, hingga sekarang algoritma ini masih digunakan oleh para pemrogram. Hal ini tentu saja tidak terlepas dari cakupannya yang luas karena sangat sukar menemukan persoalan yang tidak bisa diselesaikan dengan cara *brute force*. Bahkan, ada persoalan tertentu yang hanya dapat diselesaikan dengan algoritma *brute force*.

III. DEADLOCK

A. Pengertian Deadlock

Sebelum membahas pengertian dari *deadlock*, kita harus mengerti terlebih dahulu mekanisme sistem operasi dalam menangani sebuah proses. Proses adalah kode program yang berada dalam tahap eksekusi (*running-time*). Dalam menjalankan tugasnya, proses memerlukan satu atau lebih sumber daya (*resource*). Sumber daya yang diperlukan bisa berasal dari memori maupun alat *input/output*. Dalam keadaan normal, ada tiga tahap dalam penggunaan sumber daya oleh sebuah proses, yaitu tahap permintaan, tahap penggunaan dan tahap pelepasan [3].

Sebelum menggunakan sebuah sumber daya, proses harus mengirimkan permintaan terlebih dahulu. Terkadang, permintaan terhadap suatu sumber daya tidak bisa langsung dipenuhi karena ada proses lain yang sedang menggunakannya. Akibatnya, proses tersebut harus menunggu sampai mendapatkan sumber daya yang diperlukan. Apabila permintaan sudah dipenuhi, proses sudah memasuki tahap penggunaan. Dalam tahap ini, proses dapat menggunakan sumber daya untuk mendukung tugas/pekerjaan yang harus dilakukannya. Setelah selesai, proses harus melepaskan sumber daya yang telah dipakai agar sumber daya tersebut dapat digunakan oleh proses lain yang mungkin membutuhkannya.

Setiap proses harus melalui ketiga tahap tersebut jika ingin menggunakan sebuah sumber daya. Masalah mungkin muncul ketika sebuah proses memerlukan sebuah sumber daya yang sedang dipakai oleh proses lain. Hal ini mungkin berujung ke dalam kondisi *deadlock*. *Deadlock* adalah suatu kondisi ketika sebuah proses yang sedang menunggu permintaan untuk suatu sumber daya tidak pernah bisa mendapatkan sumber daya yang diperlukannya karena sedang dipakai oleh proses lain.

B. Karakteristik Deadlock

Dalam kondisi *deadlock*, proses tidak pernah menyelesaikan eksekusinya dan sumber daya selalu terikat kepada satu proses sehingga proses lain tidak bisa menggunakannya untuk memulai eksekusi. Ada empat karakteristik *deadlock* [3]. Karakteristik ini merupakan syarat-syarat yang memungkinkan terjadinya kondisi *deadlock*.

Petama, *mutual exclusion*. Hanya ada satu proses yang boleh menggunakan suatu sumber daya dalam satu waktu. Kondisi ini adalah suatu kebijakan (*policy*) dari sistem operasi untuk menjamin sinkronisasi proses. Kebijakan ini hanya berlaku untuk beberapa proses tertentu. Misalnya, dalam proses penulisan file. Ketika proses sedang menulis di dalam file A, proses lain yang ingin membaca data dari file A tidak akan diijinkan oleh sistem operasi. Dapat dibayangkan apabila untuk persoalan tersebut tidak ada pembatasan akses file, tentu saja hasil dari proses akan kacau dan file menjadi tidak konsisten.

Kedua, *hold and wait*. Proses mengikat paling sedikit satu sumber daya dan menunggu untuk mendapatkan sumber daya yang diperlukan selanjutnya. Maksud dari mengikat dalam konteks ini adala proses tidak melepaskan sumber daya yang telah selesai digunakannya sampai proses tersebut mendapatkan sumber daya yang diperlukannya.

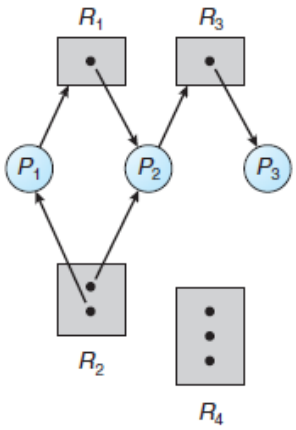
Ketiga, *no preemption*. Penggunaan sumber daya tidak bisa disela (*interrupted*). Dengan kata lain, sumber daya hanya akan tersedia apabila proses yang menggunakannya telah selesai dan melepaskannya. Tidak peduli seberapa tinggi prioritas proses yang mengirimkan permintaan, proses tersebut harus tetap menunggu sampai sumber daya tersedia.

Keempat, *circular wait*. Kondisi ini hampir menyerupai kondisi *hold and wait*. Perbedaannya, dalam *circular wait* satu proses dan proses lainnya saling terkait. Misalnya ada sekumpulan proses $\{P_0, P_1, P_2, \ldots, P_n\}$. P_0 menunggu untuk sumber daya yang sedang digunakan oleh P_1. P_1 menunggu sumber daya yang sedang digunakan oleh P_2, P_{n-1} menunggu sumber daya yang sedang digunakan oleh P_n, dan P_n menunggu sumber daya yang sedang digunakan oleh P_0.

C. *Resource-Allocation Graph*

Kondisi *deadlock* dapat digambarkan dengan lebih rinci menggunakan graf berarah yang biasa disebut graf alokasi sumber daya (*resource-allocation graph*) [3]. Seperti graf pada umumnya, graf ini terdiri atas sisi dan simpul. Dalam graf alokasi ini, simpul dikategorikan menjadi dua jenis yang berbeda, yaitu simpul P dan simpul R. Simpul P

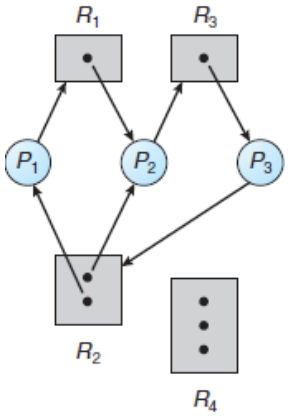
merepresentasikan himpunan proses {P_0, P_1, P_2, …, P_n} sedangkan simpul R melambangkan himpunan sumber daya dalam sistem {R_1, R_2, R_3, …, R_m}. Sisi berarah dari graf, merepresentasikan hubungan antara proses dan sumber daya. Sisi yang menunjuk dari arah P menuju R (P → R) menunjukkan bahwa proses P mengirimkan permintaan kepada sumber daya R. Sisi ini biasa disebut *request edge*. Sebaliknya, sisi yang menunjuk dari arah R menuju P (R → P) menunjukkan bahwa sumber daya R sedang dipakai oleh proses P. Sisi ini biasa disebut *assignment edge*. Untuk gambaran yang lebih jelas, dapat dilihat gambar di bawah ini.



Gambar 1.

Graf Alokasi Sumber Daya

Berikut penjelasan mengenai gambar tersebut. P_1 sedang menggunakan R_2 dan mengirim permintaan untuk menggunakan R_1. P_2 sedang menggunakan R_1 dan R_2 serta mengirim permintaan untuk menggunakan R_3. P_3 sedang menggunakan R_3. Dari penggambaran tersebut, dapat disimpulkan bahwa *deadlock* tidak akan terjadi karena setiap proses pasti akan mendapatkan sumber daya yang diperlukannya. Kondisi yang berbeda akan terjadi apabila untuk suatu waktu graf menggambarkan hal seperti ini



Gambar 2.

Graf Alokasi Sumber Daya dengan *deadlock*

Gambar di atas menunjukkan bahwa P1, P2 , dan P3 berada dalam kondisi *deadlock*. Hal ini terjadi karena P2 menunggu untuk R3 yang sedang dipakai P3. P3 juga menunggu baik P1 ataupun P2 untuk melepaskan R2. Di samping itu, P1 menunggu P2 untuk melepaskan R1. Hal ini mengindikasikan bahwa *deadlock* dapat terjadi ketika di dalam sistem terjadi sebuah siklus. Dari gambar tersebut, setidaknya ada dua siklus yang terbentuk. Siklus pertama melibatkan P_1, P_2, P_3, R_1, R_2, dan R_3 ($P_1 \rightarrow R_1 P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$). Siklus kedua melibatkan P_2, P_3, R_2, dan R_3 ($P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$). *Deadlock* tidak hanya terjadi untuk kasus seperti ini saja. Contoh di dalam gambar 1 juga bisa mengalami *deadlock* apabila setiap sumber daya hanya memiliki satu instan (*instance*). Dalam graf tersebut, instan setiap sumber daya dilambangkan dengan tanda titik.

Singkatnya, dapat disimpulkan bahwa deadlock tidak akan terjadi ketika graf alokasi sumber daya tidak menunjukkan adanya sebuah siklus. Sebaliknya, deadlock kemungkinan akan terjadi jika ada siklus. Dalam hal ini, pemrogram tidak bisa yakin sepenuhnya deadlock akan terjadi karena ketidakmampuan pemrogram untuk memantau dan melacak hal yang sebenarnya terjadi di dalam sistem pada saat *real-time*.

D. *Penanganan Deadlock*

Sehubungan dengan kondisi deadlock, ada tiga hal yang dapat dilakukan sistem operasi [3]. Pertama, sistem operasi dapat menerapkan sebuah protokol untuk menghindari atau mencegah terjadinya deadlock. Cara ini akan memastikan bahwa sistem tidak akan pernah mengalami kondisi deadlock. Kedua, sistem operasi dapat membiarkan sistem mengalami kondisi deadlock. Namun, harus ada algoritma yang bisa mendeteksinya dan sekaligus memperbaikinya. Ketiga, sistem operasi dapat mengabaikannya dan menganggap bahwa sistem tidak pernah mengalami *deadlock*. Alternatif ketiga adalah alternatif yang paling sering digunakan, misalnya dalam sistem operasi Linux dan Windows.

Seperti yang disebutkan dalam alternatif pertama, sistem operasi dapat memastikan sistem tidak mengalami kondisi *deadlock* dengan melakukan tindakan pencegahan atau penghindaran. Dengan tindakan pencegahan, sistem operasi harus menerapkan metode-metode supaya salahsatu syarat terjadinya *deadlock* tidak pernah terjadi. Hal ini biasa dilakukan dengan cara mengatur mekanisme pengiriman permintaan dari suatu proses ke sumber daya. Di sisi lain, sistem operasi akan mencatat semua sumber daya yang diperlukan oleh suatu proses selama waktu eksekusinya dalam usaha penghindaran *deadlock*. Dengan informasi tambahan tersebut, diharapkan sistem dapat memutuskan apakah permintaan suatu proses bisa langsung dipenuhi atau proses tersebut harus menunggu terlebih dahulu. Berikut ini adalah penjelasan mengenai tindakan untuk mencegah dan menghindari terjadinya *deadlock*.

(1) Pencegahan (*Deadlock Prevention*)

Prinsip dari metode ini adalah tidak membiarkan keempat syarat penyebab *deadlock* dialami oleh sistem [3]. Dengan begitu, terjadinya deadlock dapat dicegah. Keempat syarat tersebut masing-masing harus ditangani dengan cara yang berbeda.

Pertama, untuk kondisi *mutual exclusion*. Prinsip ini

biasa digunakan oleh sumber daya yang ekslusif (*non-shareable*). Pada dasarnya, sumber daya yang tidak bersifat ekslusif (*shareable*) tidak pernah terlibat jika terjadi *deadlock*. Tindakan yang dapat dilakukan adalah mengabaikan sifat ekslusif dengan cara mengizinkan satu atau lebih proses mengakses sumber daya tersebut dalam satu waktu. Namun, aturan ini tidak dapat diabaikan begitu saja karena beberapa sumber daya memang tidak bisa diakses secara bersamaan oleh beberapa proses. Dengan argumen tersebut, dapat disimpulkan bahwa pencegahan terhadap kondisi *mutual exclusion* tidak dapat dilakukan.

Kedua, untuk kondisi *hold and wait*. Ada dua jenis metode yang dapat digunakan untuk memastikan sistem tidak pernah mengalami kondisi *hold and wait*. Metode pertama mengharuskan proses untuk mengirimkan permintaan untuk semua sumber daya yang akan diperlukan dan mengalokasikan semuanya sebelum memulai eksekusi. Lain halnya dengan metode kedua. Metode ini hanya memperbolehkan proses untuk mengirimkan permintaan sumber daya ketika proses tersebut tidak sedang menggunakan sumber daya apapun.

Baik metode pertama maupun metode kedua memiliki dua kelemahan utama. Kelemahan pertama adalah rendahnya utilisasi sumber daya (*resource utilization*). Hal ini disebabkan oleh urutan dan intensitas penggunaan sumber daya. Bisa jadi sebuah sumber daya hanya dipakai sekali dan hanya diperlukan di akhir eksekusi. Dengan situasi seperti ini, seharusnya sumber daya tersebut bisa digunakan terlebih dulu oleh proses lain. Namun, hal itu tidak mungkin terjadi karena sebuah proses harus mengalokasikan semua sumber daya yang diperlukannya sebelum melakukan eksekusi. Kelemahan kedua adalah tingginya kemungkinan terjadi *starvation*. *Starvation* terjadi ketika sebuah proses harus terus-menerus menunggu untuk mendapatkan sumber daya yang diperlukan. Hal ini disebabkan sumber daya yang diperlukan selalu dialokasi terlebih dahulu oleh proses lain yang prioritasnya lebih tinggi.

Ketiga, untuk kondisi *no preemption*. Prinsip utama dari metode yang digunakan adalah sebuah proses yang permintaannya tidak bisa langsung dipenuhi harus melepaskan semua sumber daya yang telah dialokasikannya. Secara singkat, cara kerjanya adalah seperti ini. Proses A mengirimkan permintaan untuk sumber daya. Jika tersedia, permintaan dapat langsung dipenuhi dan sumber daya dapat dialokasikan. Jika tidak tersedia, periksa apakah ada proses lain yang sedang menunggu permintaannya untuk dipenuhi dan mengalokasikan sumber daya yang diperlukan proses A. Jika ada, proses tersebut harus melepaskan semua sumber daya yang dialokasikannya. Jika tidak ada, proses A harus menunggu. Dengan begitu, proses A mungkin harus melepaskan sumber daya yang dialokasikannya jika ada proses lain yang mengirim permintaan untuk sumber daya yang ada dalam daftar alokasi proses A.

Keempat, untuk kondisi *circular wait*. Prinsip utama dari metode yang digunakan adalah membuat sebuah aturan urutan akses terhadap sumber daya. Untuk setiap sumber daya, sistem memberikan nomor yang unik. Sebagai contoh, terdapat suatu himpunan sumber daya $\{R_1, R_2, R_3,\ldots,R_m\}$. Sebuah proses bisa mengirimkan permintaan untuk sebuah sumber daya R_j jika $F(R_j) > F(R_i)$. Misalnya, *disk drive* diberi nomor 5 dan *printer* diberi nomor 10. Jika sebuah proses ingin memakai kedua sumber daya tersebut bersamaan, proses itu harus mengalokasikan dulu disk drive (R_5) baru mengalokasikan printer (R_{10}). Dengan kata lain, sebuah proses hanya bisa mendapatkan R_j setelah melepaskan alokasi R_i di mana $F(R_i) > F(R_j)$. *Circular wait* tidak akan terjadi apabila kedua aturan tersebut dipenuhi. Implikasi tersebut dapat dibuktikan dengan pemisalan seperti ini. Terdapat himpunan proses $\{P_1, P_2, P_3,\ldots,P_n\}$. Dalam kasus ini, P_i menunggu giliran untuk menggunakan R_i yang sedang dialokasi oleh $P_{i+1.}$ Karena P_{i+1} mengalokasi R_i selagi menunggu untuk mendapatkan R_{i+1}, harus berlaku aturan $F(R_i) < F(R_{i+1})$ untuk seluruh nilai i. Aturan tersebut akan menghasilkan rangkaian $F(R_0) < F(R_1) < \ldots < F(R_n) < F(R_0)$. Berdasarkan prinsip transitivitas, akan didapat kondisi $F(R_0) < F(R_0)$ yang menurut aturan tidak mungkin terjadi. Oleh karena itu, kondisi *circular wait* tidak akan pernah terpenuhi.

(2) Penghindaran (*Deadlock Avoidance*)

Metode pertama mencegah terjadinya *deadlock* dengan memberikan batasan terhadap cara suatu proses dalam mengirimkan permintaan. Batasan tersebut menjamin paling tidak salahsatu dari empat syarat penyebab *deadlock* tidak terjadi. Bila dibandingkan dengan metode tersebut, metode kedua berusaha menghindari terjadinya *deadlock* dengan mencari informasi tambahan berkaitan dengan proses yang sedang berlangsung [3].

Algoritma untuk menghindari terjadinya deadlock cukup bervarisasi. Salahsatu algoritma yang cukup efektif dan paling sederhana adalah mencatat jumlah maksimum sumber daya yang diperlukan oleh setiap proses. Dengan informasi tersebut, sistem operasi bisa mengatur status dari alokasi sumber daya (*resource-allocation state*) agar tidak terjadi sebuah *circular wait*. Status alokasi sumber daya terdiri atas jumlah sumber daya yang tersedia, sumber daya yang sedang dialokasikan, dan maksimum sumber daya yang diperlukan oleh setiap proses.

Sistem akan terhindar dari kondisi *deadlock* jika memenuhi status aman (*safe state*). Status aman adalah sebuah keadaan ketika sistem mampu mengalokasikan seluruh sumber daya kepada setiap proses dengan urutan tertentu dan tidak mengalami *deadlock*. Status aman tidak pernah mengalami *deadlock*. *Deadlock* merupakan status tidak aman (*unsafe state*). Namun, tidak semua status tidak aman akan mengalami *deadlock*.

Dengan menggunakan konsep status aman, dapat didefinisikan sebuah algoritma yanag dapat memastikan sistem terhindar dari kondisi *deadlock*. Algoritma ini akan menjaga sistem untuk selalu berada di dalam status aman.

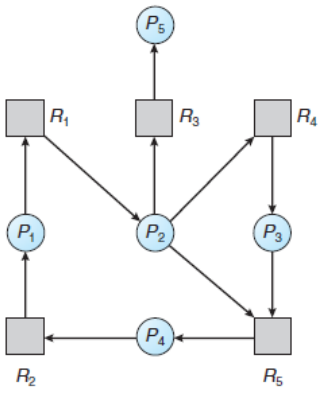
Kapanpun sebuah proses mengirim permintaan utnuk sumber daya, algoritma akan membantu sistem operasi memutuskan jika sumber daya bisa langsung dialokasikan atau proses tersebut harus menunggu. Permintaan hanya akan dipenuhi jika setelah sunmber daya dialokasikan sistem tetap berada di dalam status aman.

Salahsatu algoritma yang sering digunakan untuk menangani hal ini adalah algoritma *banker's*. Nama *banker* dipilih karena algoritma ini dapat digunakan dalam sebuah sistem perbankan sedemikian rupa sehingga kebutuhan semua nasabah masih dapat terpenuhi meskipun bank mengeluarkan sejumlah uang. Penjelasan rinci dari algoritma *banker's* tidak akan dibahas dalam karya tulis ini.

E. Pendeteksian *Deadlock*

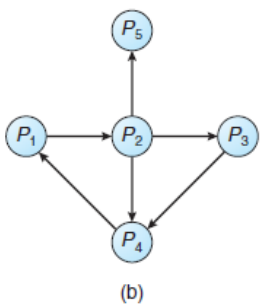
Apabila sebuah sistem operasi tidak menerapkan mekanisme untuk mencegah maupun menghindari terjadinya *deadlock*, tentu saja *deadlock* bisa terjadi. Ada beberapa hal yang dapat dilakukan oleh sistem untuk situasi seperti ini. Sistem operasi dapat menjalankan sebuah algoritma untuk memeriksa status sistem dan memberi kembalian jika terjadi *deadlock*. Setelah itu, sistem operasi dapat menjalankan algoritma lain yang berfungsi untuk memulihkan sistem dari kondisi *deadlock* [3].

Sebelum mendeteksi terjadi atau tidaknya *deadlock*, status dari sistem harus diketahui terlebih dahulu. Status ini dapat divisualisasikan lebih jelas dengan menggunakan *wait-for graph*. Graf ini merupakan bentuk modifikasi dari graf alokasi sumber daya. Berikut ini adalah ilustrasinya



Gambar 3

(a) Graf Alokasi Sumber Daya



(b) *wait-for graph*

Dapat dilihat bahwa wait-for graph menghilangkan semua simpul yang melambangkan sumber daya dalam sistem. Sisi-sisi yang terhubung juga ikut dihilangkan. Dengan demikian, sisi berarah dalam *wait-for graph* menjelaskan sebuah proses yang mengirim permintaan untuk mengalokasikan sumber daya yang sedang digunakan oleh proses lain.

Sama seperti sebelumnya, kondisi *deadlock* akan terjadi apabila terdapat siklus permintaan dalam sistem. Begitu pula dengan graf ini. Untuk itu, sistem perlu memanggil algoritma untuk memeriksa keberadaan siklus dalam *wait-for graph*. Pemeriksaan biasanya dilakukan untuk interval waktu tertentu.

IV. ANALISIS MASALAH

Seperti yang telah disebutkan dalam penjelesan mengenai *deadlock*, sistem operasi yang tidak mengimplementasikan pencegahan maupun penghindaran *deadlock* tetap harus mampu mendeteksi terjadinya *deadlock* di dalam sistem. Sehubungan dengan pendeteksian tersebut, penulis akan mencoba menggunakan pendekatan *brute force* untuk membangun algoritmanya.

Prinsip dasar dari algoritma ini adalah memeriksa setiap proses dan semua permintaaan sumber daya yang dikirimkan oleh proses tersebut. Algoritma ini akan menggunakan *wait-for graph* sebagai acuan untuk pemeriksaan. Algoritma akan mengunjungi setiap simpul dan mencatat sisi-sisi yang terhubung dengan simpul tersebut

Sebagai contoh, dapat ditinjau *wait-for graph* yang terdapat dalam Gambar 3(b). Simpul akan diperiksa secara terurut berdasarkan nomor yang tercantum. Pertama, simpul P_1 akan dikunjungi. Setelah itu, setiap sisi yang berasal dari P_1 dicatat. Dalam kasus ini hanya terdapat satu sisi, yaitu sisi dari P_1 ke P_2. Selanjutnya, P_2 akan dikunjungi dan dua sisi yang berasal dari P_2 dicatat. Pemeriksaan dilakukan dengan cara yang sama untuk semua simpul. Proses akan terus berlanjut dan berhenti ketika semua simpul sudah dikunjungi.

Sisi yang terdapat dalam graf dapat direpresentasikan sebagai nilai *boolean*. Fungsi akan mengembalikan 1 (*true*) jika terdapat sisi dari satu simpul ke simpul lainnya dan 0 (*false*) jika tidak ada sisi yang menghubungkan simpul tersebut ke simpul tertentu. Supaya lebih mudah, data ditampilkan dengan menggunakan tabel.

	P_1	P_2	P_3	P_4	P_5
P_1		0	0	1	0
P_2	1		0	0	0
P_3	0	1		0	0
P_4	0	1	1		0
P_5	0	1	0	0	

Tabel 1. Representasi data sistem berpotensi deadlock

Kolom di dalam tabel melambangkan simpul asal dari sisi sedangkan baris di dalam table menunjukkan simpul tujuan dari sisi. Misalnya, untuk baris pertama kolom kedua tabel. Sisi yang digambarkan adalah sisi dari simpul P_1 menuju simpul P_2. Sel berisi angka 1 jika sisi yang bersesuaian ada di dalam graf dan 0 jika tidak ada.

Dengan melihat data di dalam tabel, dapat ditentukan jika sistem berpotensi mengalami *deadlock*. Penentuan ini didasarkan pada terdapat atau tidaknya siklus dalam graf. Di dalam tabel, siklus akan terdeteksi ketika baris dan kolom dengan nama yang sama memiliki angka 1 di dalam salahsatu selnya. Tabel 1 menunjukkan bahwa sistem mengandung siklus yang melibatkan P_1, P_2, P_3, dan P_4. Keputusan ini didapatkan karena angka 1 terdapat di dalam sel (P_1,P_4), (P_2,P_1), (P_3,P_2), (P_4,P_2), (P_5,P_2), dan (P_4,P_3).

Siklus dalam sistem dapat dihilangkan dengan menghapus sisi yang berasal dari P_3 ke P_4 dan menghapus sisi yang berasal dari P_4 ke P_1. Perubahan tersebut mengakibatkan perbedaan data di dalam tabel. Berikut ini adalah perubahannya.

	P_1	P_2	P_3	P_4	P_5
P_1		0	0	0	0
P_2	1		0	0	0
P_3	0	1		0	0
P_4	0	1	0		0
P_5	0	1	0	0	

Tabel 2. Representasi data sistem aman

Penghapusan kedua sisi yang dilakukan akan menggantikan angka 1 di dalam sel (P_3,P_4) dan (P_4,P_1) dengan angka 0. Dengan pergantian tersebut, kolom P_3 dan kolom P_4 tidak lagi memiliki sel yang berisi angka 1. Siklus tidak akan terjadi jika hanya salahsatu di antara baris dan kolom dengan nama sama memiliki sel berisi angka 1. Sekarang, sistem sudah aman karena tidak adanya siklus.

Sekarang mungkin muncul pertanyaan mengenai penggunaan dari algoritma ini. Sebenarnya, jawaban dari pertanyaan itu bergantung terhadap sistem operasi. Kondisi yang menjadi pertimbangan adalah seberapa sering *deadlock* terjadi dan seberapa banyak sumber daya yang terpengaruh oleh *deadlock* tersebut. Jika *deadlock* sering terjadi, semakin sering pula pemanggilan algoritma pendeteksi ini. Namun, harus diingat bahwa setiap pemanggilan akan berpengaruh terhadap utilitas CPU. Jadi, ada baiknya jika algoritma ini dipanggil untuk suatu interval yang tetap atau hanya dipanggil ketika CPU mengindikasikan utilitas yang rendah.

V. KESIMPULAN

Algoritma pendeteksian *deadlock* bisa dilakukan melalui pendekatan *brute force*. Kompleksitas dari algoritma tersebut adalah $O(n^2)$ dengan n adalah jumlah simpul. Melalui analisis tersebut dapat dibuktikan bahwa algoritma brute force memenga dapat digunakan untuk menyelsaikan sebagian besar persoalan komputasi meskipun kinerjanya kurang optimal.

REFERENSI

[1] http://dictionary.reference.com/browse/brute+force.

[2] Rinaldi Munir, *Diktat Kuliah Strategi Algoritma IF2211*.Bandung: ITB, 2009, ch. 1

[3] A.Silberschatz, P.B.Galvin, G.Gagne, *Operating System Concepts Ninth Edition*. New York:John Wiley & Sons, 2013, pp. 317–333.

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, 19 Mei 2014

ttd

Rafi Ramadhan - 13512075