Bab 3

Sinkronisasi dan Deadlock

Tujuan Pelajaran

Setelah mempelajari bab ini, Anda diharapkan :

- Memahami konsep dasar sinkronisasi dalam kaitannya dengankerjasama diantara proses-proses yang ada
- Memahami bagaimana proses penanganan maslalah pada proses yang secara bersamaan menggunakan memori yang sama
- Memahami konsep deadlock, apa saja dapat menimbulkan deadlock dan bagaimana mengatasinya

3.1. Sinkronisasi

Bab ini membicarakan proses-proses untuk saling berkordinasi. Bab ini juga akan menjawab pertanyaan-pertanyaan seperti, bagaimana proses bekerja dengan sumber daya yang dibagi-bagi.

Bagaimana memastikan hanya ada satu proses yang mengakses memori pada suatu saat? Bagaimana sinkronisasi benar-benar digunakan?

3.1.1. Latar Belakang

- Akses-akses yang dilakukan secara bersama-sama ke data yang sama, dapat menyebabkan data menjadi tidak konsisten.
- Untuk menjaga agar data tetap konsisten, dibutuhkan mekanisme-mekanisme untuk memastikan pemintaan ekseskusi dari proses yang bekerja.
- *Race Condition*: Situasi dimana beberapa proses mengakses dan memanipulasi data secara bersamaan. Nilai terakhir dari data bergantung dari proses mana yang selesai terakhir.
- Untuk menghindari *Race Condition*, proses-proses secara bersamaan harus disinkronisasikan.

3.1.1.1. Kasus Produsen-Konsumer

Dua proses berbagi sebuah buffer dengan ukuran yang tetap. Salah satunya produser, meletakkan informasi ke buffer yang lainnya. Konsumen mengambil informasi dari buffer. Ini juga dapat digeneralisasi untuk masalah yang memiliki m buah produsen dan n buah konsumen, tetapi kita hanya akan memfokuskan kasus dengan satu produsen dan satu konsumen karena diasumsikan dapat menyederhanakan solusi.

Masalah akan timbul ketika produsen ingin menaruh barang yang baru tetapi buffer sudah penuh. Solusi untuk produsen adalah istirahat (*sleep*) dan akan dibangunkan ketika konsumen telah mengambil satu atau lebih barang dari buffer. Biasanya jika konsumen ingin mengambil barang dari buffer dan melihat bahwa buffer sedang kosong, maka konsumen istirahat (*sleep*) sampai produsen meletakkan barang pada buffer dan membangunkan (*wake up*) consumer.

Pendekatan seperti ini terdengar cukup sederhana, tetapi hal ini dapat menggiring kita ke jenis masalah yang sama seperti *race condition* dengan spooler direktori.

Untuk mengetahui jumlah barang di buffer, kita membutuhkan sebuah variabel kita namakan count. Jika jumlah maksimum dairi barang yang dapat ditampung buffer adalah N, kode produser pertama kali akan mencoba untuk mengetahui apakah nilai count sama dengan nilai N. Jika itu terjadi maka produsen akan istirahat (*sleep*), tetapi jika nilai count tidak sama dengan N, produsen akan terus menambahkan barang dan menaikkan nilai count.

Sekarang mari kita kembali ke permasalahan race condition. Ini dapat terjadi karena akses ke count tidak dipaksakan. Situasi seperti itu mungkin dapat terjadi. Buffer sedang kosong dan konsumen baru saja membaca count untuk melihat apakah count bernilai 0. Pada saat itu, penjadual memutuskan untuk mengentikan proses konsumen sementara dan menjalakan produsen. Produsen memasukkan barang ke buffer, menaikkan nilai count, dan memberitahukan bahwa count sekarang bernilai 1. Pemikiran bahwa count baru saja bernilai 0 sehingga konsumen harus istirahat (*sleep*). Produsen memanggil fungsi *wake up* untuk membangkitkan konsumen.

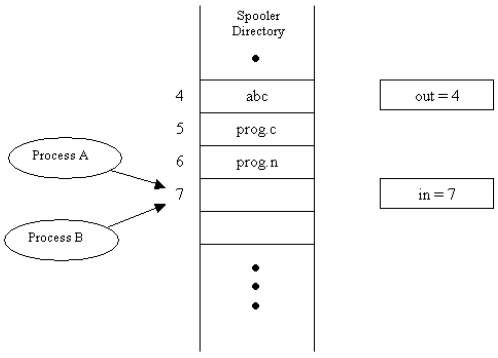
Sayangnya, konsumen secara logika belum istirahat. Jadi sinyal untuk membangkitkan konsumen, tidak dapat ditangkap oleh konsumen. Ketika konsumen bekerja berikutnya, konsumen akan memeriksa nilai count yang dibaca sebelumnya, dan mendapatkan nilai 0, kemudian konsumen istirahat (*sleep*) lagi. Cepat atau lambat produsen akan mengisi buffer dan juga pergi istirahat (*sleep*). Keduanya akan istirahat selamanya.

Inti permasalahannya disini adalah pesan untuk membangkitkan sebuah proses tidak tersampaikan. Jika pesan/ sinyal ini tersampaikan dengan baik, segalanya akan berjalan lancar.

3.1.1.2. *Race Condition*

Race Condition adalah situasi di mana beberapa proses mengakses dan memanipulasi data bersama pada saat besamaan. Nilai akhir dari data bersama tersebut tergantung pada proses yang terakhir selesai. Unutk mencegah *race condition*, proses-proses yang berjalan besamaan haus di disinkronisasi. Dalam beberapa sistem operasi, proses-proses yang berjalan bersamaan mungkin untuk membagi beberapa penyimpanan umum, masing-masing dapat melakukan proses baca (*read*) dan proses tulis (*write*). Penyimpanan bersama (*shared storage*) mungkin berada di memori utama atau berupa sebuah berkas bersama, lokasi dari memori bersama tidak merubah kealamian dari komunikasi atau masalah yang muncul. Untuk mengetahui bagaimana komunikasi antar proses bekerja, mari kita simak sebuah contoh sederhana, sebuah print spooler. Ketika sebuah proses ingin mencetak sebuah berkas, proses tersebut memasukkan nama berkas ke dalam sebuah spooler direktori yang khusus. Proses yang lain, printer daemon, secara periodik memeriksa untuk mengetahui jika ada banyak berkas yang akan dicetak, dan jika ada berkas yang sudah dicetak dihilangkan nama berkasnya dari direktori.

Bayangkan bahwa spooler direktori memiliki slot dengan jumlah yang sangat besar, diberi nomor 0, 1, 2, 3, 4,... masing-masing dapat memuat sebuah nama berkas. Juga bayangkan bahwa ada dua variabel bersama, out, penunjuk berkas berikutnya untuk dicetak, dan in, menunjuk slot kosong di direktori. Dua vaiabel tersebut dapat menamgami sebuah two-word berkas untuk semua proses. Dengan segera, slot 0, 1, 2, 3 kosong (berkas telah selesai dicetak), dan slot 4, 5, 6 sedang terisi (berisi nama dari berkas yang antre untuk dicetak). Lebih atau kurang secara besamaan, proses A dan B, mereka memutuskan untuk antre untuk sebuah berkas untuk dicetak. Situasi seperti ini diperlihatkan oleh Gambar 3-1.



Gambar 3-1. Race Condition.

Dalam *Murphy's Law* kasus tesebut dapat terjadi. Proses A membaca in dan menyimpan nilai "7" di sebuah variabel lokal yang disebut next_free_slot. Sebuah clock

interrupt terjadi dan CPU memutuskan bahwa proses A berjalan cukup lama, sehingga digantika oleh proses B. Proses B juga membaca in, dan juga mengambil nilai 7, sehingga menyimpan nama berkas di slot nomor 7 dan memperbaharui nilai in menjadi 8. Maka proses mati dan melakukan hal lain.

Akhirnya proses A berjalan lagi, dimulai dari tempat di mana proses tersebut mati. Hal ini terlihat dalam next_free_slot, ditemukan nilai 7 di sana, dan menulis nama berkas di slot nomor 7, menghapus nama berkas yang bau saja diletakkan oleh proses B. Kemudian proses A menghitung next_free_slot + 1, yang nilainya 8 dan memperbaharui nilai in menjadi 8. Direktori spooler sekarang secara internal konsisten, sehingga printer daemon tidak akan memberitahukan apa pun yang terjadi, tetapi poses B tidak akan mengambil output apa pun. Situasi seperti ini, dimana dua atau lebih proses melakukan proses reading atau writing beberapa shared data dan hasilnya bergantung pada ketepatan berjalan disebut *race condition*.

3.1.2. *Critical Section*

Bagaimana menghindari *race conditions*? Kunci untuk mencegah masalah ini dan di situasi yang lain yang melibatkan shared memori, shared berkas, and shared sumber daya yang lain adalah menemukan beberapa jalan untuk mencegah lebih dari satu proses untuk melakukan proses writing dan reading kepada shared data pada saat yang sama. Dengan kata lain kita memutuhkan *mutual exclusion*, sebuah jalan yang menjamin jika sebuah proses sedang menggunakan shared berkas, proses lain dikeluarkan dari pekerjaan yang sama. Kesulitan yang terjadi karena proses 2 mulai menggunakan variabel bersama sebelum proses 1 menyelesaikan tugasnya.

Masalah menghindari *race conditions* dapat juga diformulasikan secara abstrak. Bagian dari waktu, sebuah proses sedang sibuk melakukan perhitungan internal dan hal lain yang tidak enggiring ke kondisi *race conditions*. Bagaimana pun setiap kali sebuah proses mengakses shared memory atau shared berkas atau melakukan sesuatu yang kitis akan menggiring kepada *race conditions*. Bagian dari program dimana shaed memory diakses disebut *Critical Section* atau *Critical Region*.

Walau pun dapat mencegah *race conditions*, tapi tidak cukup untuk melakukan kerjasama antar proses secara pararel dengan baik dan efisien dalam menggunakan shared data. Kita butuh 4 kondisi agar menghasilkan solusi yang baik:

I. Tidak ada dua proses secara bersamaan masuk ke dalam citical section.
II. Tidak ada asumsi mengenai kecepatan atau jumlah cpu.
III. Tidak ada proses yang berjalan di luar critical secion yang dapat mengeblok proses lain.
IV. Tidak ada proses yang menunggu selamamya untuk masuk critical section.

Critical Section adalah sebuah segmen kode di mana sebuah proses yang mana sumber daya bersama diakses. Terdiri dari: *Entry Section*: kode yang digunakan untuk masuk ke dalam *critical section*

Critical Section: Kode di mana hanya ada satu proses yang dapat dieksekusi pada satu waktu
Exit Section: akhir dari *critical section*, mengizinkan proses lain
Remainder Section: kode istirahat setelah masuk ke *critical section*

Solusi yang diberikan harus memuaskan permintaaan berikut:

- *Mutual exclution*
- *Deadlock free*

- *Starvation free*

Pendekatan yang mungkin untuk solusi proses sinkronisasi

i. Solusi Piranti lunak (Software solution)
- Tanpa Sinkronisasi.
- Dengan Sinkronisasi.
- Low-level primitives: *semaphore*
- High-level primitives: *monitors*

ii. Solusi Piranti Keras (Hardware solution)

3.1.2.1. *Mutual Exclusion*

Mutual Exclusion: Kondisi-kondisi untuk solusi

Tiga kondisi untuk menentukan *mutual Exclusion*

1. Tidak ada dua proses yang pada saat bersamaan berada di *critical region*.
2. Tidak ada proses yang berjalan diluar *critical region* yang bisa menghambat proses lain
3. Tidak ada proses yang tidak bisa masuk ke *critical region*

3.1.2.2. Solusi

Cara-cara memecahkan masalah

• Hanya dua proses, Po dan P1
• Struktur umum dari proses adalah Pi (proses lain Pj)

```
do {
    critical section
    remainder section
} while(1);
```

Gambar 3-2. Critical Section.

3.1.2.2.1. Algoritma 1

Disini kita akan mencoba membuat sebuah rangkaian solusi-solusi dari permasalahan yang makin meningkat kerumitannya.
Pada semua contoh, i adalah proses yang sedang berjalan, j adalah proses yang lain. Pada contoh ini *code*.

i. Shared variables
- int turn
  Initially turn=0
- turn = i, *Pi can enter its critical section*

ii. Process Pi

```
do {
    while(turn!=1);
    critical section
    turn=j;
    remainder section
} while(1);
```

Gambar 3-3. Proses Pi.

iii. Memenuhi *mutual exclusion*, tapi bukan progress.

3.1.2.2.2. Algoritma 2

FLAG untuk setiap proses yang memberi STATE:

Setiap proses memantau suatu flag yang mengindikasikan ia ingin memasuki critical section. Dia memeriksa flag poses lain dan tidak akan memasuki critical section bila ada proses lain yang sedang masuk.

i. Shared variables
- boolean flag[2];
  initially flag [0] = flag [1] = false

- flag [i] = true , *Pi ready to enter its critical section*

ii. Process Pi

```
do {
    flag[i]:=true;
    while(turn!=1);
    critical section
    turn=j;
    remainder section
} while(1);
```

Gambar 3-4. Process Pi.

iii. Memenuhi *mutual exclusion*, tapi tidak memenuhi progess.

3.1.2.2.3. Algoritma 3

FLAG untuk meminta izin masuk:

- Setiap proses mengeset sebuah flag untuk meminta izin masuk. Lalu setiap proses mentoggle bit untuk mengizinkan yang lain untuk yang pertama

- Kode ini dijalankan untuk setiap proses i

```
Shared variables
F boolean flag[2];
initially flag[0] = flag[1] = false
F flag[i] = true;
```

Gambar 3-5. Kode.

Pi ready to enter its critical section

- Gabungan shared variables dari algorima 1 dan 2

- Process Pi

```
do {
    flag[i]:=true;
    turn = j;
    while(flag[j] and turn = j);
    critical section
```

```
        flag[i] = false;
        remainder section
} while(1);
```

Gambar 3-6. Process Pi.

- Memenuhi ketiga persyaratan, memecahkan persoalan *critical section* untuk kedua proses

3.1.2.2.4. Algoritma *Bakery*

Critical Section untuk n buah proses:

Sebelum memasukkan proses ke *critical section*, proses menerima sebuah nomor. Pemegang nomor terkecil masuk ke *critical section*. Jika ada dua proses atau lebih menerima nomor sama, maka proses dengan indeks terkecil yang dilayani terlebih dahulu untuk masuk ke critical section. Skema penomoran selalu naik secara berurut contoh: 1, 2, 3, 3, 3, 3, 4, 5,...

```
boolean choosing [n];
long long long int number [n];
/* 64 bit maybe okay for about 600 years */
Array structure elements are initiallized to false and 0 respectively
while (true) {
        choosing[i] = true;
        number[i] = max(number[0], ... [n-1]) + 1;
        choosing[i] = false;
        for (j = 0; j < n; j ++) {
            while (choosing[j]) {}
            while ((number[j] !=0) && ((number[j], j) < (number[i], i))) {}
        }
        number[i] = 0
}
Solves the critical-section problem
for n process
```

Gambar 3-7. Process Pi.

3.1.3. Solusi Hardware pada Sinkronisasi

Disabling Interrupts: Hanya untuk uni prosesor saja.

Atomic test and set: Returns parameter and sets parameter to true atomically.

```
while (test_and_set(lock));
/* critical section */
lock = false;
GET_LOCK: IF_CLEAR_THEN_SET_BIT_AND_SKIP (bit_address)
BRANCH GET_LOCK /* set failed */
                    /* set succeeded */
```

Gambar 3-8. Process Pi.

Harus hati-hati jika pendekatan ini untuk menyelesaikan *bounded-buffer* - harus menggunakan *round robin* - memerlukan kode yang dibuat di sekitar instruksi *lock*.

```
while (test_and_set(lock));
```

```
Boolean waiting[N];
int j; /* Takes on values from 0 to N - 1 */
Boolean key;
do {
        waiting[i] = TRUE;
        key = TRUE;
        while ( waiting[i] && key )
                key = test_and_set( lock ); /* Spin lock */
        waiting[i] = FALSE;

    /****** CRITICAL SECTION ********/
    j = ( i + 1 ) mod N;
    while ( ( j != i ) && ( ! waiting[ j ] ) )
    j = ( j + 1 ) % N;
    if ( j == i ) //Using Hardware
            lock = FALSE; //Test_and_set.
    else
            waiting[ j ] = FALSE;
    /******* REMAINDER SECTION *******/
} while (TRUE);
```

Gambar 3-9. Lock.

3.1.4. *Semaphore*

Jika kita ingin dapat melakukan proses tulis lebih rumit kita membutuhkan sebuah bahasa untuk melakukannya. Kita akhirnya medefinisikan semaphore yang kita asumsikan sebagai sebuah operasi atomik.

Semaphore adalah pendekatan yang diajukan oleh Djikstra, dengan prinsip bahwa dua proses atau lebih dapat bekerja sama dengan menggunakan penanda-penanda sederhana. Seperti proses dapat dipaksa berhenti pada suatu saat, sampai proses mendapatkan penanda tertentu itu. Sembarang kebutuhan koordinasi kompleks dapat dipenuhi dengan struktur penanda yang cocok untuk kebutuhan itu. Variabel khusus untuk penanda ini disebut semaphore.

Semaphore mempunyai dua sifat, yaitu:

i. Semaphore dapat diinisialisasi dengan nilai non-negatif.
ii. Terdapat dua operasi terhadap semaphore, yaitu Down dan Up. Usulan asli yang disampaikan Djikstra adalah operasi P dan V.

3.1.4.1. Operasi *Down*

Operasi ini menurunkan nilai semaphore, jika nilai semaphore menjadi non-positif maka proses yang mengeksekusinya diblocked.

```
Type Semaphore = Integer,

Procedure Down(Var: semaphore);
Begin

        s := s-1;
        if s <= 0 Then
        Begin
                Tempatkan antrian pada antrian untuk semaphore s
        Proses diblocked
```

```
        End;
End;
```

Gambar 3-10. Block.

Operasi Down adalah atomic, tak dapat diinterupsi sebelaum diselesaikan.Emnurunkan nilai, memeriksa nilai, menempatkan proses pada antrian dan memblocked sebagai instruksi tunggal. Sejak dimulai, tak ada proses alain yang dapat mengakses semaphore sampai operasi selesai atau diblocked.

3.1.4.2. Operasi *Up*

Operasi Up menakkan nilai semaphore. Jika satu proses atau lebih diblocked pada semaphore itu tak dapat menyelesaikan operasi Down, maka salah satu dipilih oleh system dan menyelesaikan operasi Down-nya. Urutan proses yang dipilih tidak ditentukan oleh Djikstra, dapat dipilih secara acak.

```
Type Semaphore = Integer,

Procedure Down(Var: semaphore);
Begin

        s := s + 1;
        if s <= 0 Then
        Begin
                Pindahkan satu proses P dari antrian untuk semaphore s
                Tempatkan proses P di senarai ready
        End;
End;
```

Gambar 3-11. Block.

Adanya semaphore mempermudah persoalan mutual exclusion. Skema penelesaian mutual exclusion mempunyai bagan sebagai berikut:

```
Cons N = 2;
Var S:semaphore;
Procedure enter_critical_section;
{
        mengerjakan kode-kode kritis
}

Procedure enter_noncritical_section;
{
        mengerjakan kode-kode tak kritis
}

ProcedureProses(i: integer);
Begin
        Repeat
        Down(s);
        Enter_critical_section;

        Up(s);
        Enter_noncritical_section;
        Forever
End;

Begin
    S:= 1;
    Parbegin
```

```
        Proses(0);
        Proses(1);
    ParEnd
End;
```

Gambar 3-12. Mutex.

Sebelum masuk *critical section*, proses melakukan Down. Bila berhasil maka proses masuk ke *critical section*. Bila tidak berhasil maka proses di-blocked atas semaphore itu. Proses yang diblocked akan dapat melanjutkan kembali bila proses yang ada di *critical section* keluar dan melakukan opersai up sehingga menjadikan proses yang diblocked ready dan melanjutkan sehingga opersi Down-nya berhasil.

3.1.5. Problem Klasik pada Sinkronisasi

Ada tiga hal yang selalu memjadi masalah pada proses sinkronisasi:

i. Problem *Bounded buffer*.
ii. Problem *Reades and Writer*.
iii. Problem *Dining Philosophers*.

3.1.5.1. Problem *Readers-Writers*

Problem lain yang terkenal adalah readers-writer problem yang memodelkan proses yang mengakses database. Sebagai contoh sebuah sistem pemesanan sebuah perusahaan penerbangan, dimana banyak proses berkompetisi berharap untuk membaca (*read*) dan menulis (*write*). Hal ini dapat diterima bahwa banyak proses membaca database pada saat yang sama, tetapi jika suatu proses sedang menulis database, tidak boleh ada proses lain yang mengakses database tersebut, termasuk membaca database tersebut.

Dalam solusi ini, pertama-tama pembaca mengakses database kemudian melakukan DOWN pada semaphore db.. Langkah selanjutnya readers hanya menaikkkan nilai sebuah counter. Hasil dari pembaca nilai counter diturunkan dan nilai terakhir dilakukan UP pada semaphore, mengizinkan memblok writer.

Misalkan selama sebuah reader menggunakan database, reader lain terus berdatangan. Karena ada dua reader pada saat bersamaan bukanlah sebuah masalah, maka reader yang kedua diterima, reader yang ketiga juga dapat diterima jika terus berdatangan reader-reader baru.

Sekarang misalkan writer berdatangan terus menerus. Writer tidak dapat diterima ke database karena writer hanya bisa mengakses data ke database secara ekslusif, jadi writer ditangguhkan. Nanti penambahan reader akan menunjukkan peningkatan. Selama paling tidak ada satu reader yang aktif, reader berikutnya jika datang akan diterima.

Sebagai konsekuensi dari strategi ini, selama terdapat suplai reader yang terus-menerus, mereka akan dilayani segera sesuai kedatanga mereka. Writer akan ditunda sampai tidak ada reader lagi. Jika sebuah reader baru tiba, katakan, setiap dua detik, dan masing-masing reader mendapatkan lima detik untuk melakukan tugasnya, writer tudak akan pernah mendapatkan kesempatan.

Untuk mencegah situasi seperti itu, program dapat ditulis agak sedikit berbeda: Ketika reader tiba dan writer menunggu, reader ditunda dibelakang writer yang justru diterima dengan segera. Dengan cara ini, writer tidak harus menunggu reader yang

sedang aktif menyelesaikan pekerjaannya, tapi tidak perlu menunggu reader lain yang datang berturut-turut setelah itu.

3.1.5.2. Problem *Dining Philosopers*

Pada tahun 1965, Djikstra menyelesaikan sebuah masalah sinkronisasi yang beliau sebut dengan dining philisophers problem. Dining philosophers dapat diuraikan sebagai berikut: Lima orang filosuf duduk mengelilingi sebuah meja bundar. Masing-masing filosof mempunyai sepiring spageti. Spageti-spageti tersebut sangat licin dan membutuhkan dua garpu untuk memakannya. Diantara sepiring spageti terdapat satu garpu.

Kehidupan para filosof terdiri dari dua periode, yaitu makan atau berpikir. Ketika seorang filosof lapar, dia berusaha untuk mendapatkan garpu kiri dan garpu kanan sekaligus. Jika sukses dalam mengambil dua garpu, filosof tersebut makan untuk sementara waktu, kemudian meletakkan kedua garpu dan melanjutkan berpikir.

Pertanyaan kuncinya adalah, dapatkah anda menulis program untuk masing-masing filosof yang melakukan apa yang harus mereka lakukan dan tidak pernah mengalami kebuntuan.

Prosedur take-fork menunggu sampai garpu-garpu yang sesuaididapatkan dan kemudian menggunakannya. Sayangnya dari solusi ini ternyata salah. Seharusnya lima orang filosof mengambil garpu kirinya secara bersamaan. Tidak akan mungkin mereka mengambil garpu kanan mereka, dan akan terjadi deadlock.

Kita dapat memodifikasi program sehingga setelah mengambil garpu kiri, program memeriksa apakah garpu kanan meungkinkan untuk diambil. Jika garpu kanan tidak mungkin diambil, filosof tersebut meletakkan kembali garpu kirinya, menunggu untuk beberapa waktu, kemudia mengulangi proses yang sama. Usulan tersebut juga salah, walau pun dengan alasan yang berbeda. Dengan sedikit nasib buruk, semua filosof dapat memulai algoritma secara bersamaan, mengambil garpu kiri mereka, melihat garpu kanan mereka yang tidak mungkin untuk diambil, meletakkan kembali garpu kiri mereka, menunggu, mengambil garpu kiri mereka lagi secara bersamaan, dan begitu seterusnya. Situasi seperti ini dimana semua program terus berjalan secara tidak terbatas tetapi tidak ada perubahan/kemajuan yang dihasilkan disebut starvation.

Sekarang anda dapat berpikir "jika filosof dapat saja menunggu sebuah waktu acak sebagai pengganti waktu yang sama setelah tidak dapat mengambil garpu kiri dan kanan, kesempatan bahwa segala sesuatau akan berlanjut dalam kemandegan untuk beberapa jam adalah sangat kecil." Pemikiran seperti itu adalah benar,tapi beberapa aplikasi mengirimkan sebuah solusi yang selalu bekerja dan tidak ada kesalahan tidak seperti hsk nomor acak yang selalu berubah.

Sebelum mulai mengambil garpu, seorang filosof melakukan DOWN di mutex. Setelah menggantikan garpu dia harus melakukan UP di mutex. Dari segi teori, solusi ini cukup memadai. Dari segi praktek, solusi ini tetap memiliki masalah. Hanya ada satu filosof yang dapat makan spageti dalam berbagai kesempatan. Dengan lima buah garpu, seharusnya kita bisa menyaksikan dua orang filosof makan spageti pada saat bersamaan.

Solusi yang diberikan diatas benar dan juga mengizinkan jumlah maksimum kegiatan paralel untuk sebuah jumlah filosf yang berubah-ubah ini menggunakan sebuah array, state, untuk merekam status seorang filosof apakah sedang makan (*eating*), berpikir (*think*), atau sedang lapar (*hungry*) karena sedang berusaha mengambil garpu. Seorang

filosof hanya dapat berstatus makan (*eating*) jika tidak ada tetangganya yang sedang makan juga. Tetangga seorang filosof didefinisikan ole LEFT dan RIGHT.

Dengan kata lain, jika i = 2, maka tetangga kirinya (LEFT) = 1 dan tetangga kanannya (RIGHT) = 3. Program ini menggunakan sebuah array dari semaphore yang lapar (hungry) dapat ditahan jika garpu kiri atau kanannya sedang dipakai tetangganya. Catatan bahwa masing-masing proses menjalankan prosedur filosof sebagai kode utama, tetapi prosedur yang lain seperti take-forks, dan test adalah prosedur biasa dan bukan proses-proses yang terpisah.

3.1.6. Monitors

Solusi sinkronisasi ini dikemukakan oleh Hoare pada tahun 1974. Monitor adalah kumpulan prosedur, variabel dan struktur data di satu modul atau paket khusus. Proses dapat memanggil prosedur-prosedur kapan pun diinginkan. Tapi proses tak dapat mengakses struktur data internal dalam monitor secara langsung. Hanya lewat prosedur-prosedur yang dideklarasikan minitor untuk mengakses struktur internal.

Properti-properti monitor adalah sebagai berikut:

i. Variabel-variabel data lokal, hanya dapat diakses oleh prosedur-prosedur dala monitor dan tidak oleh prosedur di luar monitor.

ii. Hanya satu proses yang dapat aktif di monitor pada satu saat. Kompilator harus mengimplementasi ini(mutual exclusion).

iii. Terdapat cara agar proses yang tidak dapat berlangsung di-blocked. Menambahkan variabel-variabel kondisi, dengan dua operasi, yaitu Wait dan Signal.

iv. ait: Ketika prosedur monitor tidak dapat berlanjut (misal producer menemui buffer penuh) menyebabkan proses pemanggil diblocked dan mengizinkan proses lain masuk monitor.

v. Signal: Proses membangunkan partner-nya yang sedang diblocked dengan signal pada variabel kondisi yang sedang ditunggu partnernya.

vi. Versi Hoare: Setelah signal, membangunkan proses baru agar berjalan dan menunda proses lain. vii. Versi Brinch Hansen: Setelah melakukan signal, proses segera keluar dari monitor.

Dengan memaksakan disiplin hanya satu proses pada satu saat yang berjalan pada monitor, monitor menyediakan fasilitas mutual exclusion. Variabel-variabel data dalam monitor hanya dapat diakses oleh satu proses pada satu saat. Struktur data bersama dapat dilindungi dengan menempatkannya dalam monitor. Jika data pada monitor merepresentasikan sumber daya, maka monitor menyediakan fasilitas mutual exclusion dalam mengakses sumber daya itu.

3.2. *Deadlock*

Pada pembahasan di atas telah dikenal suatu istilah yang populer pada bagian *semaphores*, yaitu *deadlock*. Secara sederhana *deadlock* dapat terjadi dan menjadi hal yang merugikan, jika pada suatu saat ada suatu proses yang memakai sumber daya dan ada proses lain yang menunggunya. Bagaimanakah *deadlock* itu yang sebenarnya? Bagaimanakah cara penanggulangannya?

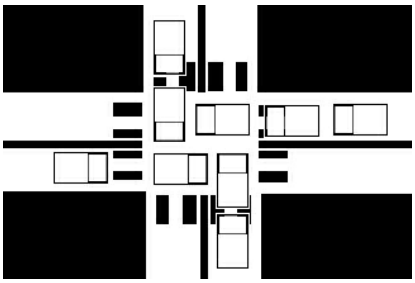
3.2.1. Latar Belakang

Misalkan pada suatu komputer terdapat dua buah program, sebuah *tape drive* dan sebuah *printer*. Program A mengontrol *tape* drive, sementara program B mengontrol *printer*. Setelah beberapa saat, program A meminta *printer*, tapi *printer* masih digunakan. Berikutnya, B meminta *tape drive*, sedangkan A masih mengontrol *tape drive*. Dua program tersebut memegang kontrol terhadap sumber daya yang dibutuhkan oleh program yang lain. Tidak ada yang dapat melanjutkan proses masing-masing sampai program yang lain memberikan sumber dayanya, tetapi tidak ada yang mengalah. Kondisi inilah yang disebut *Deadlock* atau pada beberapa buku disebut *Deadly Embrace Deadlock* yang mungkin dapat terjadi pada suatu proses disebabkan proses itu menunggu suatu kejadian tertentu yang tidak akan pernah terjadi. Dua atau lebih proses dikatakan berada dalam kondisi *deadlock*, bila setiap proses yang ada menunggu suatu kejadian yang hanya dapat dilakukan oleh proses lain dalam himpunan tersebut.

Terdapat kaitan antara *overhead* dari mekanisme koreksi dan manfaat dari koreksi *deadlock* itu sendiri. Pada beberapa kasus, *overhead* atau ongkos yang harus dibayar untuk membuat sistem bebas deadlock menjadi hal yang terlalu mahal dibandingkan jika mengabaikannya. Sementara pada kasus lain, seperti pada *real-time process control*, mengizinkan *deadlock* akan membuat sistem menjadi kacau dan membuat sistem tersebut tidak berguna.

Contoh berikut ini terjadi pada sebuah persimpangan jalan. Beberapa hal yang dapat membuat *deadlock* pada suatu persimpangan, yaitu:

- Terdapat satu jalur pada jalan.
- Mobil digambarkan sebagai proses yang sedang menuju sumber daya.
- Untuk mengatasinya beberapa mobil harus *preempt* (mundur).
- Sangat memungkinkan untuk terjadinya *starvation* (kondisi proses tak akan mendapatkan sumber daya).

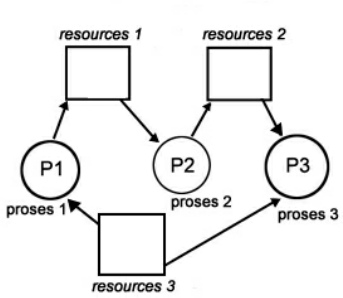


Gambar 3-13. Persimpangan.

3.2.2. *Resources-Allocation Graph*

Sebuah cara visual (matematika) untuk menentukan apakah ada *deadlock*, atau kemungkinan terjadinya. G = (V, E) Graf berisi *node and edge*. *Node* V terdiri dari proses-proses = {P1, P2, P3, ...} dan jenis *resource*. {R1, R2, ...} *Edge* E adalah (Pi, Rj) atau (Ri, Pj) Sebuah panah dari *process* ke *resource* menandakan proses meminta *resource*.

Sebuah panah dari *resource* ke *process* menunjukkan sebuah *instance* dari *resource* telah dtempatkan ke proses. *Process* adalah lingkaran, *resource* adalah kotak; titik-titik merepresentasikan jumlah *instance* dari *resource* Dalam tipe. Meminta poin-poin ke kotak, perintah datang dari titik.

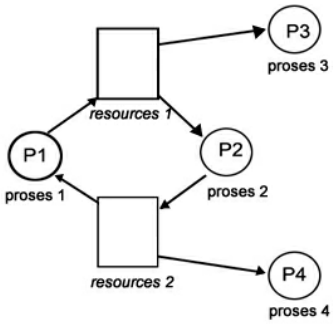


Gambar 3-14. Graph.

Jika graf tidak berisi lingkaran, maka tidak ada proses yang *deadlock*.

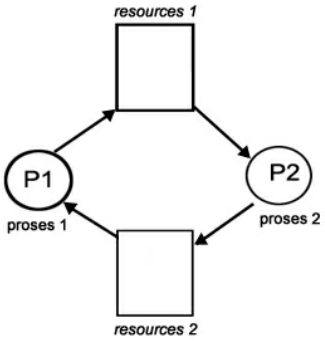
Jika membentuk lingkaran, maka:

i. jika tipe *resource* memiliki banyak *instance*, maka *deadlock* DAPAT ada.



Gambar 3-15. Non Deadlock.

ii. jika setiap tipe *resource* mempunyai satu *instance*, maka *deadlock* telah terjadi.



Gambar 3-16. Deadlock.

3.2.3. Model Sistem

Menurut *Coffman* dalam bukunya *"Operating System"* menyebutkan empat syarat bagi terjadinya *deadlock*, yaitu:

i. Mutual Exclusion
Suatu kondisi dimana setiap sumber daya diberikan tepat pada satu proses pada suatu waktu.

ii. Hold and Wait
Kondisi yang menyatakan proses-proses yang sedang memakai suatu sumber daya dapat meminta sumber daya yang lain.

iii. Non-pre-emptive
Kondisi dimana suatu sumber daya yang sedang berada pada suatu proses tidak dapat diambil secara paksa dari proses tersebut,sampai proses itu melepaskannya.

iv. Circular Wait
Kondisi yang menyatakan bahwa adanya rantai saling meminta sumber daya yang dimiliki oleh suatu proses oleh proses lainnya.

3.2.4. Strategi menghadapi *Deadlock*

Strategi untuk menghadapi *deadlock* dapat dibagi menjadi tiga pendekatan, yaitu:

i. Mengabaikan adanya *deadlock*.

ii. Memastikan bahwa *deadlock* tidak akan pernah ada, baik dengan metode. Pencegahan, dengan mencegah empat kondisi *deadlock* agar tidak akan pernah terjadi. Metode Menghindari *deadlock*, yaitu mengizinkan empat kondisi *deadlock*, tetapi menghentikan setiap proses yang kemungkinan mencapai *deadlock*.

iii. Membiarkan *deadlock* untuk terjadi, pendekatan ini membutuhkan dua metode yang saling mendukung, yaitu:

• Pendeteksian *deadlock*, untuk mengidentifikasi ketika *deadlock* terjadi.
• Pemulihan *deadlock*, mengembalikan kembali sumber daya yang dibutuhkan pada proses yang memintanya.

Dari penjabaran pendekatan diatas, terdapat empat metode untuk mengatasi *deadlock* yang akan terjadi,
yaitu:

3.2.4.1. Strategi *Ostrich*

Pendekatan yang paling sederhana adalah dengan menggunakan strategi burung unta: masukkan kepala dalam pasir dan seolah-olah tidak pernah ada masalah sama sekali. Beragam pendapat muncul berkaitan dengan strategi ini. Menurut para ahli Matematika, cara ini sama sekali tidak dapat diterima dan semua keadaan *deadlock* harus ditangani. Sementara menurut para ahli Teknik, jika komputer lebih sering mengalami kerusakkan disebabkan oleh kegagalan *hardware*, *error* pada kompilator atau *bugs* pada sistem operasi. Maka ongkos yang dibayar untuk melakukan penanganan

deadlock sangatlah besar dan lebih baik mengabaikan keadaan *deadlock* tersebut. Metode ini diterapkan pada sistem operasi UNIX dan MINIX.

3.2.5. Mencegah *Deadlock*

Metode ini merupakan metode yang paling sering digunakan. Metode Pencegahan dianggap sebagai solusi yang bersih dipandang dari sudut tercegahnya *deadlock*. Tetapi pencgahan akan mengakibatkan kinerja utilisasi sumber daya yang buruk.

Metode pencegahan menggunakan pendekatan dengan cara meniadakan empat syarat yang dapatmenyebabkan deadlock terjadi pada saat eksekusi Coffman (1971).

Syarat pertama yang akan dapat ditiadakan adalah *Mutual Exclusion*, jika tidak ada sumber daya yang secara khusus diperuntukkan bagi suatu proses maka tidak akan pernah terjadi *deadlock*. Namun jika membiarkan ada dua atau lebih proses mengakses sebuah sumber daya yang sama akan menyebabkan *chaos*. Langkah yang digunakan adalah dengan *spooling* sumber daya, yaitu dengan mengantrikan *job-job* pada antrian dan akan dilayani satu-satu.

Beberapa masalah yang mungkin terjadi adalah:

i. Tidak semua dapat di-*spool*, tabel proses sendiri tidak mungkin untuk di-*spool*
ii. Kompetisi pada ruang disk untuk *spooling* sendiri dapat mengarah pada *deadlock*

Hal inilah yang menyebabkan mengapa syarat pertama tidak dapat ditiadakan, jadi *mutual exclusion* benar-benar tidak dapat dihilangkan.

Cara kedua dengan meniadakan kondisi *hold and wait* terlihat lebih menjanjikan. Jika suatu proses yang sedang menggunakan sumber daya dapat dicegah agar tidak dapat menunggu sumber daya yang lain, maka *deadlock* dapat dicegah. Langkah yang digunakan adalah dengan membuat proses agar meminta sumber daya yang mereka butuhkan pada awal proses sehingga dapat dialokasikan sumber daya yang dibutuhkan. Namun jika terdapat sumber daya yang sedang terpakai maka proses tersebut tidak dapat memulai prosesnya.

Masalah yang mungkin terjadi:

i. Sulitnya mengetahui berapa sumber daya yang dibutuhkan pada awal proses

ii. Tidak optimalnya pengunaan sumber daya jika ada sumber daya yang digunakan hanya beberapa waktu dan tidak digunakan tapi tetap dimiliki oleh suatu proses yang telah memintanya dari awal.

Meniadakan syarat ketiga *non preemptive* ternyata tidak lebih menjanjikan dari meniadakan syarat kedua, karena dengan meniadakan syarat ketiga maka suatu proses dapat dihentikan ditengah jalan. Hal ini tidak dimungkinkan karena hasil dari suatu proses yang dihentikan menjadi tidak baik.

Cara terakhir adalah dengan meniadakan syarat keempat *circular wait*. Terdapat dua pendekatan, yaitu:

i. Mengatur agar setiap proses hanya dapat menggunakan sebuah sumber daya pada suatu waktu, jika menginginkan sumber daya lain maka sumber daya yang dimiliki harus dilepas.
ii. Membuat penomoran pada proses-proses yang mengakses sumber daya. Suatu proses dimungkinkan untuk dapat meminta sumber daya kapan pun, tetapi permintaannya harus dibuat terurut.

Masalah yang mungkin terjadi dengan mengatur bahwa setiap proses hanya dapat memiliki satu proses adalah bahwa tidak semua proses hanya membutuhkan satu sumber daya, untuk suatu proses yang kompleks dibutuhkan banyak sumber daya pada saat yang bersamaan. Sedangkan dengan penomoran masalah yang dihadapi adalah tidak terdapatnya suatu penomoran yang dapat memuaskan semua pihak.

Secara ringkas pendekatan yang digunakan pada metode pencegahan deadlock dan masalah-masalah yang menghambatnya, terangkum dalam tabel dibawah ini.

Tabel 3-1. Tabel Deadlock

Syarat	Langkah	Kelemahan
Mutual Exclusion	*Spooling* sumber daya	Dapat menyebabkan *chaos*
Hold and Wait	Meminta sumber daya di awal	Sulit memperkirakan di awal dan tidak optimal
No Pre-emptive	Mengambil sumber daya di tengah proses	Hasil proses tidak akan baik
Circular Wait	Penomoran permintaan sumber daya	Tidak ada penomoran yang memuaskan semua pihak

3.2.6. Menghindari *Deadlock*

Pendekatan metode ini adalah dengan hanya memberi kesempatan ke permintaan sumber daya yang tidak mungkin akan menyebabkan *deadlock*. Metode ini memeriksa dampak pemberian akses pada suatu proses, jika pemberian akses tidak mungkin menuju kepada *deadlock*, maka sumber daya akan diberikan pada proses yang meminta. Jika tidak aman, proses yang meminta akan di-*suspend* sampai suatu waktu permintaannya aman untuk diberikan. Kondisi ini terjadi ketika setelah sumber daya yang sebelumnya dipegang oleh proses lain telah dilepaskan.

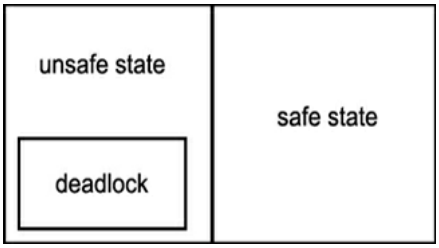
Kondisi aman yang dimaksudkan selanjutnya disebut sebagai *safe-state*, sedangkan keadaan yang tidak memungkinkan untuk diberikan sumber daya yang diminta disebut *unsafe-state*.

3.2.6.1. Kondisi Aman (*Safe state*)

Suatu keadaan dapat dinyatakan sebagai *safe state* jika tidak terjadi *deadlock* dan terdapat cara untuk memenuhi semua permintaan sumber daya yang ditunda tanpa menghasilkan *deadlock*. Dengan cara mengikuti urutan tertentu.

3.2.6.2. Kondisi Tak Aman (*Unsafe state*)

Suatu *state* dinyatakan sebagai *state* tak selamat (*unsafe state*) jika tidak terdapat cara untuk memenuhi semua permintaaan yang saat ini ditunda dengan menjalankan proses-proses dengan suatu urutan.



Gambar 3-17. Safe.

3.2.7. Algoritma *Bankir*

Algoritma penjadualan ini diungkapkan oleh Dijkstra (1965) lebih dikenal dengan nama Algoritma Bankir. Model ini menggunakan suatu kota kecil sebagai percontohan dengan suatu bank sebagai sistem operasi, pinjaman sebagai sumber daya dan peminjam sebagai proses yang membutuhkan sumber daya.

Deadlock akan terjadi apabila terdapat seorang peminjam yang belum mengembalikan uangnya dan ingin meminjam kembali, padahal uang yang belum dikembalikan tadi dibutuhkan oleh peminjam lain yang juga belum mengembalikan uang pinjamannya.
Beberapa kelemahan algoritma Bankir Tanenbaum (1992), Stallings (1995) dan Deitel (1990) adalah sebagai berikut:

i. Sulit untuk mengetahui seluruh sumber daya yang dibutuhkan proses pada awal eksekusi.

ii. Jumlah proses yang tidak tetap dan berubah-ubah.

iii. Sumber daya yang tadinya tersedia dapat saja menjadi tidak tersedia kembali.

iv. Proses-proses yang dieksekusi haruslah tidak dibatasi oleh kebutuhan sinkronisasi antar proses.

v. Algoritma ini menghendaki memberikan semua permintaan selama waktu yang berhingga.

3.2.8. Mendeteksi *Deadlock* dan Memulihkan *Deadlock*

Metode ini mengunakan pendekatan dengan teknik untuk menentukan apakah *deadlock* sedang terjadi serta proses-proses dan sumber daya yang terlibat dalam deadlock tersebut. Setelah kondisi *deadlock* dapat dideteksi, maka langkah pemulihan dari kondisi *deadlock* dapat segera dilakukan. Langkah pemulihan tersebut adalah dengan memperoleh sumber daya yang diperlukan oleh proses-proses yang membutuhkannya. Beberapa cara digunakan untuk mendapatkan sumber daya yang diperlukan, yaitu dengan terminasi proses dan *pre-emption* (mundur) suatu proses. Metode ini banyak digunakan pada komputer *mainframe* berukuran besar.

3.2.8.1. Terminasi Proses

Metode ini akan menghapus proses-proses yang terlibat pada kondisi *deadlock* dengan mengacu pada beberapa syarat. Beberapa syarat yang termasuk dalam metode ini adalah, sebagai berikut:

- Menghapus semua proses yang terlibat dalam kondisi *deadlock* (solusi ini terlalu mahal).

- Menghapus satu persatu proses yang terlibat, sampai kondisi *deadlock* dapat diatasi (memakan banyak waktu).

- Menghapus proses berdasarkan prioritas, waktu eksekusi, waktu untuk selesai, dan kedalaman dari *rollback*.

3.2.8.2. *Resources Preemption*

Metode ini lebih menekankan kepada bagaimana menghambat suatu proses dan sumber daya, agar tidak terjebak pada *unsafe condition*.

Beberapa langkahnya, yaitu:

• Pilih salah satu - proses dan sumber daya yang akan di-*preempt*.
• *Rollback* ke *safe state* yang sebelumnya telah terjadi.
• Mencegah suatu proses agar tidak terjebak pada *starvation* karena metode ini.

3.3. Kesimpulan

Untuk mengatasi problem critical section dapat digunakan berbagai solusi software. Namun masalah yang akan timbul dengan solusi software adalah solusi software tidak mampu menangani masalah yang lebih berat dari critical section. Tetapi Semaphores mampu menanganinya, terlebih jika hardware yang digunakan mendukung maka akan memudahkan dalam menghadapi problem sinkronisasi.

Berbagai contoh klasik problem sinkronisasi berguna untuk mengecek setiap skema baru sinkronisasi. Monitor termasuk ke dalam level tertinggi mekanisme sinkronisasi yang berguna untuk mengkoordinir aktivitas dari banyak thread ketika mengakses data melalui pernyataan yang telah disinkronisasi

Kondisi *deadlock* akan dapat terjadi jika terdapat dua atau lebih proses yang akan mengakses sumber daya yang sedang dipakai oleh proses yang lainnya. Pendekatan untuk mengatasi *deadlock* dipakai tiga buah pendekatan, yaitu:

• Memastikan bahwa tidak pernah dicapai kondisi *deadlock*
• Membiarkan *deadlock* untuk terjadi dan memulihkannya
• Mengabaikan apa pun *deadlock* yang terjadi

Dari ketiga pendekatan diatas, dapat diturunkan menjadi empat buah metode untuk mengatasi *deadlock*, yaitu:

• Pencegahan *deadlock*
• Menghindari *deadlock*
• Mendeteksi *deadlock*
• Pemulihan *deadlock*

Namun pada sebagian besar Sistem Operasi dewasa ini mereka lebih condong menggunakan pendekatan untuk mengabaikan semua *deadlock* yang terjadi Silberschatz

(1994) merumuskan sebuah strategi penanggulangan deadlock terpadu yang dapat disesuaikan dengan kondisi dan situasi yang berbeda, strateginya sendiri berbunyi:

1. Kelompokkan sumber daya kedalam kelas yang berbeda

2. Gunakan strategi pengurutan linear untuk mencegah kondisi *circular wait*yang nantinya akan mencegah deadlock diantara kelas sumber daya

3. Gunakan algoritma yang paling cocok untuk suatu kelas sumber daya yang berbeda satu dengan yang lain

3.4. Latihan

1. Proses dapat meminta berbagai kombinasi dari sumber daya dibawah ini: *CDROM*, *soundcard* dan *floppy*. Jelaskan tiga macam pencegahan deadlock skema yang meniadakan:
   - *Hold and Wait*
   - *Circular Wait*
   - *No Preemption*

2. Diasumsikan proses P0 memegang sumber daya R2 dan R3, meminta sumber daya R4; P1 menggunakan R4 dan meminta R1; P2 menggunakan R1 dan meminta R3 . Gambarkan *Wait-for Graph*. Apakah sistem terjebak dalam *deadlock*? Jika ya, tunjukkan proses mana yang menyebabkan *deadlock*. Jika tidak, tunjukkan urutan proses untuk selesai.

3. User x telah menggunakan 7 printer dan harus menggunakan 10 printer. User y telah menggunakan 1 printer dan akan memerlukan paling banyak 4 printer. User z telah menggunakan 2 printer dan akan menggunakan paling banyak 4 printer. Setiap user pada saat ini meminta 1 printer. Kepada siapakah OS akan memberikan grant printer tersebut dan tunjukkan "safe sequence" yang ada sehingga tidak terjadi deadlock.

4. Pernyataan manakah yang benar mengenai deadlock:
   i. Pencegahan deadlock lebih sulit dilakukan (implementasi) daripada menghindari deadlock.
   ii. Deteksi deadlock dipilih karena utilisasi dari resources dapat lebih optimal.
   iii. Salah satu prasyarat untuk melakukan deteksi deadlock adalah: hold and wait.
   iv. Algoritma Banker's (Djikstra) tidak dapat menghindari terjadinya deadlock.
   v. Suatu sistem jika berada dalam keadaan tidak aman: "unsafe", berarti telah terjadi deadlock.

5. User 1 sedang menggunakan x printers dan memerlukan total n printers. Kondisi umum adalah: y < -12, n < -12, x < -y, m < -n. State ini safe jika dan hanya jika:

   i. x+n<-12 dan y+m<-12 dan x+m<-12
   ii. x+n<-12 dan y+m<12 dan x+m<-12
   iii. x+n<-12 atau(or) y+m<-12 dan x+m<-12
   iv. x+m<-12
   v. Semua statement diatas menjamin: safe state

3.5. Rujukan

3.5.1. Rujukan Sinkronisasi

1. Hariyanto, B.,"*Sistem Operasi*", Bandung: Informatika, Desember 1997
2. Silberschatz, A., Gagne, G. dan Galvin, P., "*Applied Operating System Concept*", John Wiley and Sons Inc., 2000
3. Tanenbaum, Andrew S., "*Modern Operating Systems*", Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1992

3.5.2. Rujukan *Deadlock*

1. Coffman, E.G., Jr., M.J. Elphick dan A. Shoshani, "*System Deadlocks*", Computing surveys, Vol.3, No.2, June 1971
2. Deitel, H.M., "*Operating Systems*", 2nd Edition, Massachusetts: Addison-WesleyPublishing Company, 1990
3. Hariyanto, B.,"Sistem Operasi", Bandung: Informatika, Desember 1997
4. Havender, J.W., "*Avoiding Deadlock in Multitasking Systems*", IBM Systems Journal, Vol.7, No.2, 1968. *97*
5. Silberschatz, A., Gagne, G. dan Galvin, P., "*Applied Operating System Concept*", John Wiley and Sons Inc., 2000
6. Tanenbaum, Andrew S., "*Modern Operating Systems*", Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1992