MATA KULIAH SISTEM OPERASI
KODE MATA KULIAH CCS210

DISUSUN OLEH
NIZIRWAN ANWAR & TEAM

FAKULTAS ILMU KOMPUTER
UNIVERSITAS ESA UNGGUL
JAKARTA
2018

MATERI

“Thread & Deadlock”
(PENJADUALAN)

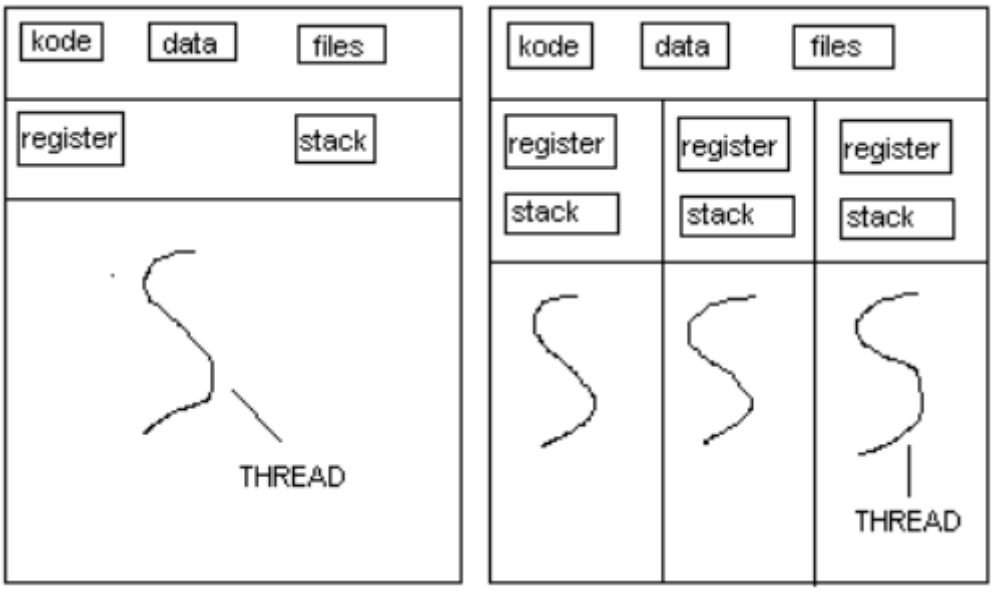
5.1 Konsep Thread dan Deadlock

5.1.1 Thread

Model proses yang didiskusikan sejauh ini telah menunjukkan bahwa suatu proses adalah sebuah program yang menjalankan eksekusi *thread* tunggal. Sebagai contoh, jika sebuah proses menjalankan sebuah program *Word Processor*, ada sebuah *thread* tunggal dari instruksi-instruksi yang sedang dilaksanakan. Kontrol *thread* tunggal ini hanya memungkinkan proses untuk menjalankan satu tugas pada satu waktu.

Banyak sistem operasi modern telah memiliki konsep yang dikembangkan agar memungkinkan sebuah proses untuk memiliki eksekusi *multi-threads*, agar dapat secara terus menerus mengetik dan menjalankan pemeriksaan ejaan didalam proses yang sama, maka sistem operasi tersebut memungkinkan proses untuk menjalankan lebih dari satu tugas pada satu waktu.

Thread merupakan unit dasar dari penggunaan CPU, yang terdiri dari Thread_ID, *program counter*, *register set*, dan *stack*. Sebuah *thread* berbagi *code section*, *data section*, dan sumber daya sistem operasi dengan Thread lain yang dimiliki oleh proses yang sama. Thread juga sering disebut *lightweight process*. Sebuah proses tradisional atau *heavyweight process* mempunyai *thread* tunggal yang berfungsi sebagai pengendali. Perbedaan antara proses dengan *thread* tunggal dengan proses dengan *thread* yang banyak adalah proses dengan *thread* yang banyak dapat mengerjakan lebih dari satu tugas pada satu satuan waktu.



Gambar Thread (a) Single (b) Multi

Banyak perangkat lunak yang berjalan pada PC modern dirancang secara *multi-threading*. Sebuah aplikasi biasanya diimplementasi sebagai proses yang terpisah dengan beberapa *thread* yang berfungsi sebagai pengendali. Contohnya sebuah *web browser* mempunyai *thread* untuk menampilkan gambar atau tulisan sedangkan *thread* yang lain berfungsi sebagai penerima data dari network.

Kadang kala ada situasi dimana sebuah aplikasi diperlukan untuk menjalankan beberapa tugas yang serupa. Sebagai contohnya sebuah *web server* dapat mempunyai ratusan klien yang mengaksesnya secara *concurrent*. Kalau *web server* berjalan sebagai proses yang hanya mempunyai *thread* tunggal maka ia hanya dapat melayani satu klien pada pada satu satuan waktu. Bila ada klien lain yang ingin mengajukan permintaan maka ia harus menunggu sampai klien sebelumnya selesai dilayani. Solusinya adalah dengan membuat *web server* menjadi *multi-threading*.

Dengan ini maka sebuah *web server* akan membuat *thread* yang akan mendengar permintaan klien, ketika permintaan lain diajukan maka *web server* akan menciptakan *thread* lain yang akan melayani permintaan tersebut. Java mempunyai pengunaan lain dari *thread*. Perlu diketahui bahwa Java tidak mempunyai konsep *asynchronous*. Sebagai contohnya kalau program java mencoba untuk melakukan koneksi ke server maka ia akan berada dalam keadaan block state sampai koneksinya jadi (dapat dibayangkan apa yang terjadi apabila servernya mati).

Karena Java tidak memiliki konsep *asynchronous* maka solusinya adalah dengan membuat *thread* yang mencoba untuk melakukan koneksi ke server dan *thread* lain yang pertamanya tidur selamabeberap waktu (misalnya 60 detik) kemudian bangun. Ketika waktu tidurnya habis maka ia akan bangun dan memeriksa apakah *thread* yang melakukan koneksi ke server masih mencoba untuk melakukan koneksi ke server, kalau *thread* tersebut masih dalam keadaan mencoba untuk melakukan koneksi ke server maka ia akan melakukan interrupt dan mencegah *thread* tersebut untuk mencoba melakukan koneksi ke server.

5.1.2 Deadlock

Deadlock dalam arti sebenarnya adalah kebuntuan. Kebuntuan yang dimaksud dalam sistem operasi adalah kebuntuan proses. Jadi *Deadlock* ialah suatu kondisi dimana proses tidak berjalan lagi atau pun tidak ada komunikasi lagi antar proses. *Deadlock* disebabkan karena proses yang satu menunggu sumber daya yang sedang dipegang oleh proses lain yang sedang menunggu sumber daya yang dipegang oleh proses tersebut. Dengan kata lain setiap proses dalam set menunggu untuk sumber yang hanya dapat dikerjakan oleh proses lain dalam set yang sedang menunggu. Contoh sederhananya ialah pada gambar berikut ini

Proses P1 Proses P2

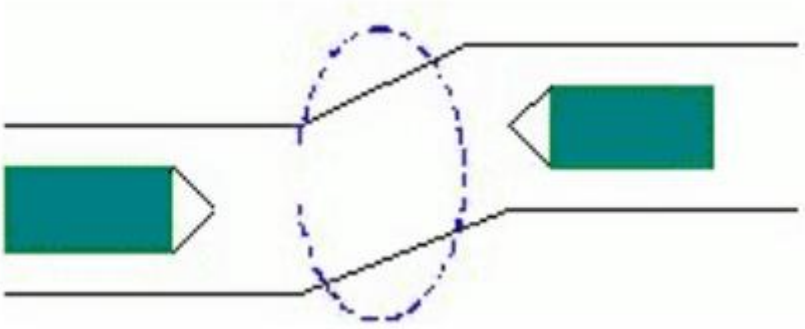
....
....

Receive (P2); Receive (P1);

....
....

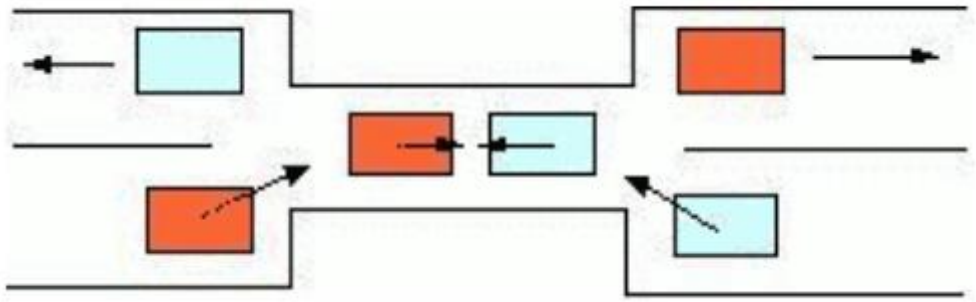
Send (P2, M1); Send (P1, M2);

Proses tersebut dapat direpresentasikan dengan gambar sebagai berikut

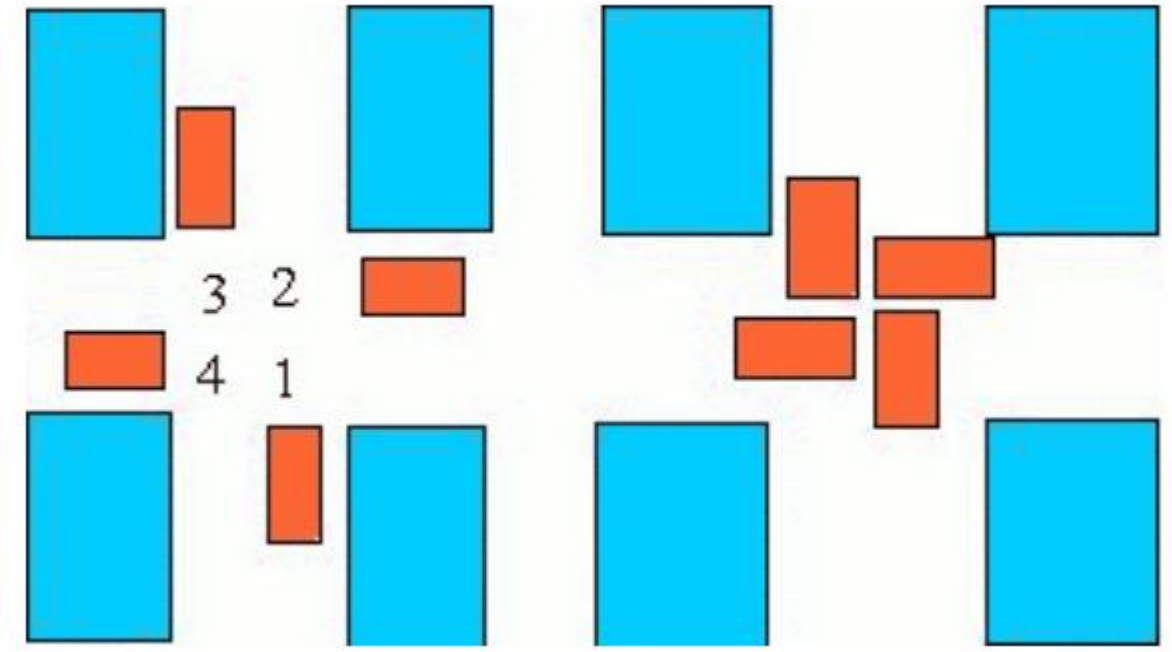


Gambar 2 Contoh Deadlock pada rel kereta
(Disadur dari www.tvcc.cc.or.us/staff/fuller/ cs160/chap3/chap3.html)

Dari gambar (2) tersebut bisa dilihat bahwa kedua kereta tersebut tidak dapat berjalan. Karena kedua kereta tersebut saling menunggu kereta yang lain untuk lewat dulu agar keretanya dapat berjalan. Sehingga terjadilah *Deadlock*.



Gambar 3 Contoh Deadlock di Jembatan Gantung
(*Modern Operating Systems*, Tanenbaum, 1992)



Gambar 4 Contoh Deadlock di Persimpangan Jalan
(*Operating Systems, Fourth Edition*", Stallings, William,Prentice Hall, 2001)

Dalam kasus ini setiap mobil bergerak sesuai nomor yang ditentukan, tetapi tanpa pengaturan yang benar, maka setiap mobil akan bertemu pada satu titik yang permanen (yang dilingkari) atau dapat dikatakan bahwa setiap mobil tidak dapat meanjutkan perjalanan lagi atau dengan kata lain terjadi *Deadlock*. Contoh lain pada proses yang secara umum terdiri dari tiga tahap, yaitu untuk meminta, memakai, dan melepaskan sumber daya yang di mintanya.

5.2 MultThreading

Keuntungan dari program yang multithreading dapat dipisah menjadi 4 (empat) kategori:

1) Responsi dalam membuat aplikasi yang interaktif menjadi multithreading dapat membuat sebuah program terus berjalan meski pun sebagian dari program tersebut diblok atau melakukan operasi yang panjang, karena itu dapat meningkatkan respons kepada pengguna. Sebagai contohnya dalam web browser yang multithreading, sebuah thread dapat melayani permintaan pengguna sementara thread lain berusaha menampilkan image.

2) Berbagi sumber daya: thread berbagi memori dan sumber daya dengan thread lain yang dimiliki oleh proses yang sama. Keuntungan dari berbagi kode adalah mengizinkan sebuah aplikasi untuk mempunyai beberapa thread yang berbeda dalam lokasi memori yang sama.

3) Ekonomi: dalam pembuatan sebuah proses banyak dibutuhkan pengalokasian memori dan sumber daya. Alternatifnya adalah dengan penggunaan thread, karena thread berbagi memori dan sumber daya proses yang memilikinya maka akan lebih ekonomis untuk membuat dan context switch thread. Akan susah untuk mengukur perbedaan waktu antara proses dan thread dalam hal pembuatan dan pengaturan, tetapi secara umum pembuatan dan pengaturan proses lebih lama dibandingkan thread. Pada Solaris, pembuatan proses lebih lama 30 kali dibandingkan pembuatan thread, dan context switch proses 5 kali lebih lama dibandingkan context switch thread.

4) Utilisasi arsitektur multiprocessor: Keuntungan dari multithreading dapat sangat meningkat pada arsitektur multiprocessor, dimana setiap thread dapat berjalan secara pararel di atas processor yang berbeda. Pada arsitektur processor tunggal, CPU menjalankan setiap thread secara bergantian tetapi hal ini berlangsung sangat cepat sehingga menciptakan ilusi pararel, tetapi pada kenyataannya hanya satu thread yang dijalankan CPU pada satu-satuan waktu (satu-satuan waktu pada CPU biasa disebut time slice atau quantum).

5.2.1 *User* dan *Kernel Threads*

1) *User Thread*

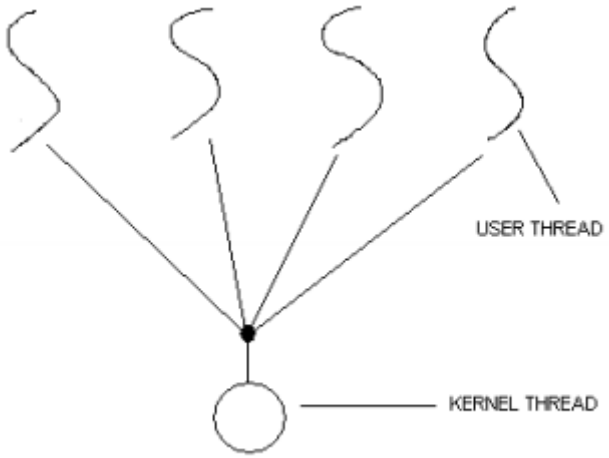
User *thread* didukung di atas kernel dan diimplementasi oleh *thread* library pada user level. *Library* menyediakan fasilitas untuk pembuatan *thread*, penjadualan *thread*, dan managemen *thread* tanpa dukungan dari kernel. Karena kernel tidak menyadari user-level *thread* maka semua pembuatan dan penjadualan *thread* dilakukan di user space tanpa intervensi dari kernel. Oleh karena itu, user-level *thread* biasanya cepat untuk dibuat dan diatur. Tetapi user *thread* mempunyai lemahan yaitu apabila kernelnya merupakan *thread* tunggal maka apabila salah satu user-level *thread* menjalankan *blocking system call* maka akan mengakibatkan seluruh proses diblok walaupun ada *thread* lain yang dapat jalan dalam aplikasi tersebut. Contoh *user-thread libraries* adalah POSIX Pthreads, Mach C-threads, dan Solaris threads.

2) *Kernel Thread*

Kernel *thread* didukung langsung oleh sistem operasi. Pembuatan, penjadualan, dan managemen *thread* dilakukan oleh kernel pada *kernel space*. Karena pengaturan *thread* dilakukan oleh sistem operasi maka pembuatan dan pengaturan kernel *thread* lebih lambat dibandingkan user *thread*. Keuntungannya adalah *thread* diatur oleh kernel, karena itu jika sebuah *thread* menjalankan *blocking system call* maka kernel dapat menjadualkan *thread* lain di aplikasi untuk melakukan eksekusi. Keuntungan lainnya adalah pada lingkungan *multiprocessor*, kernel dapat menjadual thread-thread pada processor yang berbeda. Contoh sistem operasi yang mendukung kernel *thread* adalah Windows NT, Solaris, Digital UNIX.

5.2.2 Model *MultiThreading*

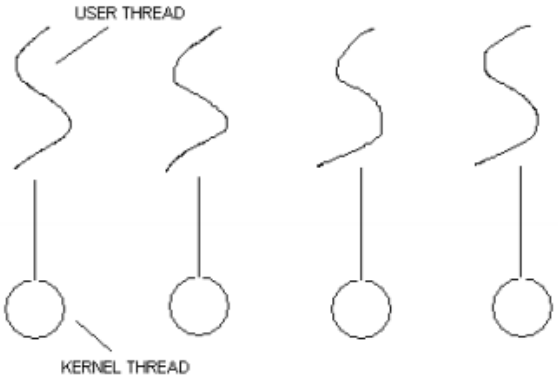
1) Many-to-One



Gambar 5 Thread Model *Many-to-One*

Many-to-One model memetakan banyak user-level *thread* ke satu kernel *thread*. Pengaturan *thread* dilakukan di *user space*, oleh karena itu ia efisien tetapi ia mempunyai kelemahan yang sama dengan user *thread*. Selain itu karena hanya satu *thread* yang dapat mengakses *thread* pada suatu waktu maka *multiple thread* tidak dapat berjalan secara pararel pada *multiprocessor*. User-level *thread* yang diimplementasi pada sistem operasi yang tidak mendukung kernel *thread* menggunakan Many-to-One model.

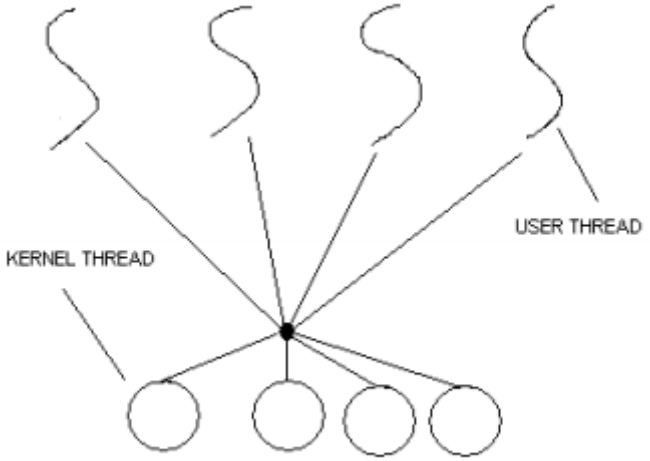
2) One-to-One



Gambar 6 Thread Model *One-to-One*

One-to-One model memetakan setiap user *thread* ke kernel *thread*. Ia menyediakan lebih banyak *concurrency* dibandingkan Many-to-One model. Keuntungannya sama dengan keuntungan kernel *thread*. Kelemahannya model ini adalah setiap pembuatan user *thread* membutuhkan pembuatan kernel *thread*. Karena pembuatan *thread* dapat menurunkan performa dari sebuah aplikasi maka implmentasi dari model ini membatasi jumlah *thread* yang dibatasi oleh sistem. Contoh sistem operasi yang mendukung One-to-One model adalah Windows NT dan OS/2.

3) Many-to-Many



Gambar 7 Thread Model *Many-to-Many*

Many-to-many model *multiplexes* banyak *user-level thread* ke kernel *thread* yang jumlahnya lebih kecil atau sama banyaknya dengan user-level *thread*. Jumlah kernel *thread* dapat spesifik untuk sebagian aplikasi atau sebagian mesin. Many-to-One model mengizinkan developer ntuk membuat user *thread* sebanyak yang ia mau tetapi *concurrency* tidak dapat diperoleh karena hanya satu *thread* yang dapat dijadual oleh kernel pada suatu waktu. One-to-One menghasilkan concurrency yang ebih tetapi developer harus hati-hati untuk tidak menciptakan terlalu banyak *thread* dalam suatu aplikasi (dalam beberapa hal, developer hanya dapat membuat *thread* dalam jumlah yang terbatas). Many-to-Many model tidak menderita kelemahan dari 2 model di atas. Developer dapat membuat user *thread* sebanyak yang diperlukan, dan kernel *thread* yang bersangkutan dapat bejalan secara pararel pada *ultiprocessor*. Dan juga ketika suatu *thread* menjalankan *blocking system call* maka kernel dapat menjadualkan *thread* lain untuk melakukan eksekusi. Contoh sistem operasi yang mendukung model ini adalah Solaris, IRIX, dan Digital UNIX.

5.3 Thread dan Proses

Thread adalah sebuah alur kontrol dari sebuah proses. Suatu proses yang multithreaded mengandung beberapa perbedaan alur kontrol dengan ruang alamat yang sama. Keuntungan dari multithreaded meliputi peningkatan respon dari pengguna, pembagian sumber daya proses, ekonomis, dan kemampuan untuk mengambil keuntungan dari arsitektur multiprosesor. *Thread* tingkat pengguna adalah thread yang tampak oleh programer dan tidak diketahui oleh kernel. *Thread* tingkat pengguna secara tipikal dikelola oleh sebuah library *thread* di ruang pengguna. *Thread* tingkat *kernel* didukung dan dikelola oleh *kernel* sistem operasi. Secara umum, *thread* tingkat pengguna lebih cepat dalam pembuatan dan pengelolaan dari pada *kernel thread*. Ada 3 perbedaan tipe dari model yang berhubungan dengan pengguna dan *kernel thread* yaitu one-to one model, many-to-one model, many-to-many model.

5.4 Kondisi untuk Terjadinya *Deadlock*

Menurut Coffman (1971) ada empat kondisi yang dapat mengakibatkan terjadinya *Deadlock*, yaitu:

1) Mutual Eksklusif: hanya ada satu proses yang boleh memakai sumber daya, dan proses lain yang ingin memakai sumber daya tersebut harus menunggu hingga sumber daya tadi dilepaskan atau tidak ada proses yang memakai sumber daya tersebut.

2) Memegang dan menunggu: proses yang sedang memakai sumber daya boleh meminta sumber daya lagi maksudnya menunggu hingga benar-benar sumber daya yang diminta tidak dipakai oleh proses lain, hal ini dapat menyebabkan kelaparan sumber daya sebab dapat saja sebuah proses tidak mendapat sumber daya dalam waktu yang lama

3) Tidak ada *Preemption*: sumber daya yang ada pada sebuah proses tidak boleh diambil begitu saja oleh proses lainnya. Untuk mendapatkan sumber daya tersebut, maka harus dilepaskan terlebih dahulu oleh proses yang memegangnya, selain itu seluruh proses menunggu dan mempersilahkan hanya proses yang memiliki sumber daya yang boleh berjalan

4) *Circular Wait*: kondisi seperti rantai, yaitu sebuah proses membutuhkan sumber daya yang dipegang proses berikutnya.

Banyak cara untuk menanggulangi *Deadlock*:

1) Mengabaikan masalah *Deadlock*.

2) Mendeteksi dan memperbaiki

3) Penghindaran yang terus menerus dan pengalokasian yang baik dengan menggunakan protokol untuk memastikan sistem tidak pernah memasuki keadaan *Deadlock*. Yaitu dengan *Deadlock* avoidance sistem untuk mendata informasi tambahan tentang proses mana yang akan meminta dan menggunakan sumber daya.

4) Pencegahan yang secara struktur bertentangan dengan empat kondisi terjadinya *Deadlock* dengan *Deadlock* prevention sistem untuk memasti- kan bahwa salah satu kondisi yang penting tidak dapat menunggu.

5.4.1 Mendeteksi dan Memperbaiki

Caranya ialah dengan cara mendeteksi jika terjadi *Deadlock* pada suatu proses maka dideteksi sistem mana yang terlibat di dalamnya. Setelah diketahui sistem mana saja yang terlibat maka diadakan proses untuk memperbaiki dan menjadikan sistem berjalan kembali. Hal-hal yang terjadi dalam mendeteksi adanya *Deadlock* adalah:

1) Permintaan sumber daya dikabulkan selama memungkinkan.

2) Sistem operasi memeriksa adakah kondisi *circular wait* secara periodik.

3) Pemeriksaan adanya *Deadlock* dapat dilakukan setiap ada sumber daya yang hendak digunakan oleh sebuah proses.

4) Memeriksa dengan algoritma tertentu.

5.4.2. Mengabaikan Masalah *Deadlock*

Metode ini lebih dikenal dengan Algoritma Ostrich. Dalam algoritma ini dikatakan bahwa untuk menghadapi *Deadlock* ialah dengan berpura-pura bahwa tidak ada masalah apa pun. Hal ini seakan-akan melakukan suatu hal yang fatal, tetapi sistem operasi Unix menanggulangi *Deadlock* dengan cara ini dengan tidak mendeteksi *Deadlock* dan membiarkannya secara otomatis mematikan program sehingga seakan-akan tidak terjadi apa pun. Jadi jika terjadi *Deadlock*, maka tabel akan penuh, sehingga proses yang menjalankan proses melalui operator harus menunggu

pada waktu tertantu dan mencoba lagi. Terdapat beberapa jalan untuk kembali dari *Deadlock*

a) **Lewat *Preemption***

Dengan cara untuk sementara waktu menjauhkan sumber daya dari pemakainya, dan memberikannya pada proses yang lain. Ide untuk memberi pada proses lain tanpa diketahui oleh pemilik dari sumber daya tersebut tergantung dari sifat sumber daya itu sendiri. Perbaikan dengan cara ini sangat sulit atau dapat dikatakan tidak mungkin. Cara ini dapat dilakukan dengan memilih korban yang akan dikorbankan atau diambil sumber dayanya utuk sementara, tentu saja harus dengan perhitungan yang cukup agar waktu yang dikorbankan seminimal mungkin. Setelah kita melakukan preemption dilakukan pengkondisian proses tersebut dalam kondisi aman. Setelah itu proses dilakukan lagi dalam kondisi aman tersebut.

b) Lewat Melacak Kembali

Setelah melakukan beberapa langkah *preemption*, maka proses utama yang diambil sumber dayanya akan berhenti dan tidak dapat melanjutkan kegiatannya, oleh karena itu dibutuhkan langkah untuk kembali pada keadaan aman dimana proses masih berjalan dan memulai proses lagi dari situ. Tetapi untuk beberapa keadaan sangat sulit menentukan kondisi aman tersebut, oleh karena itu umumnya dilakukan cara mematikan program tersebut lalu memulai kembali proses. Meski pun sebenarnya lebih efektif jika hanya mundur beberapa langkah saja sampai *Deadlock* tidak terjadi lagi. Untuk beberapa sistem mencoba dengan cara mengadakan pengecekan beberapa kali secara periodik dan menandai tempat terakhir kali menulis ke disk, sehingga saat terjadi *Deadlock* dapat mulai dari tempat terakhir penandaannya berada.

c) **Lewat membunuh proses yang menyebabkan *Deadloc*k**

Cara yang paling umum ialah membunuh semua proses yang mengalami *Deadlock*. Cara ini paling umum dilakukan dan dilakukan oleh hampir semua sistem operasi. Namun, untuk beberapa sistem, kita juga dapat membunuh beberapa proses saja dalam siklus *Deadlock* untuk menghindari *Deadlock* dan mempersilahkan proses lainnya kembali berjalan. Atau dipilih salah satu korban untuk melepaskan sumber dayanya, dengan cara ini maka masalah pemilihan korban menjadi lebih selektif, sebab telah diperhitungkan beberapa kemungkinan jika si proses harus melepaskan sumber dayanya. Kriteria seleksi korban ialah:

- Yang paling jarang memakai prosesor
- Yang paling sedikit hasil programnya
- Yang paling banyak memakai sumber daya sampai saat ini
- Yang alokasi sumber daya totalnya tersedkit
- Yang memiliki prioritas terkecil

5.5. Menghindari *Deadlock*

Pada sistem kebanyakan permintaan terhadap sumber daya dilakukan sebanyak sekali saja. Sistem sudah harus dapat mengenali bahwa sumber daya itu aman atau tidak(dalam arti tidak terkena *Deadlock*), setelah itu baru dialokasikan. Ada dua cara yaitu:

1) Jangan memulai proses apa pun jika proses tersebut akan membawa kita pada kondisi *Deadlock*, sehingga tidak mungkin terjadi *Deadlock* karena ketika akan menuju *Deadlock* sudah dicegah.
2) Jangan memberi kesempatan pada suatu proses untuk meminta sumber daya lagi jika penambahan ini akan membawa kita pada suatu keadaan *Deadlock*

Jadi diadakan dua kali penjagaan, yaitu saat pengalokasian awal, dijaga agar tidak *Deadlock* dan ditambah dengan penjagaan kedua saat suatu proses meminta sumber daya, dijaga agar jangan sampai terjadi *Deadlock*. Pada *Deadlock* avoidance sistem dilakukan dengan cara memastikan bahwa program memiliki maksimum permintaan. Dengan kata lain cara sistem ini memastikan terlebih dahulu bahwa sistem akan selalu dalam kondisi aman. Baik mengadakan permintaan awal atau pun saat meminta permintaan sumber daya tambahan, sistem harus selalu berada dalam kondisi aman.

Latihan Soal

1. Sebutkan dua perbedaan antara user level thread dan kernel thread. Saat kondisi bagaimana salah satu dari thread tersebut lebih baik

2. Jelaskan tindakan yang diambil oleh sebuah kernel saat alih konteks antara kernel level thread.

3. Sebutkan 5 (lima) aktivitas sistem operasi yang merupakan contoh dari suatu managemen proses.

4. Definisikan perbedaan antara penjadualan short term, medium term dan long term.

5. Definisikan perbedaan antara penjadualan secara preemptive dan nonpreemptive!

6. Jelaskan mengapa penjadualan strict nonpreemptive tidak seperti yang digunakan di sebuah komputer pusat.

7. Tunjukkan dua contoh pemrograman dari multithreading yang dapat meningkatkan sebuah solusi thread tunggal.

8. Tunjukkan dua contoh pemrograman dari multithreading yang tidak dapat meningkatkan sebuah solusi thread tunggal.

9. Jelaskan tentang keempat hal yang menyebabkan *deadlock*? Dan bagaimana cara mengatasi keempat masalah tersebut?

10. Pernyataan manakah yang benar mengenai deadlock:

    (a) Pencegahan deadlock lebih sulit dilakukan (implementasi) daripada menghindari deadlock.
    (b) Deteksi deadlock dipilih karena utilisasi dari resources dapat lebih optimal.
    (c) Salah satu prasyarat untuk melakukan deteksi deadlock adalah: hold and wait.
    (d) Algoritma Banker’s (Djikstra) tidak dapat menghindari terjadinya deadlock.
    (e) Suatu sistem jika berada dalam keadaan tidak aman: "*unsafe*", berarti telah terjadi deadlock.

Daftar Pustaka

Modern Operating System 4th Edition Andrew S Tanembaun 2015

Operating System, Internals and design Principles, William Stallings 6th Edition 2008

Avi Silberschatz, Peter Galvin, Greg Gagne. Applied Operationg System Concepts 1st Ed. 2000. John Wiley & Sons, Inc.