



MODUL IX CCS 210 SISTEM OPERASI

Judul	Mutual Exclusion dan Deadlock	
Penyusun	Distribusi	Perkuliahan
Nixon Erzed	FASILKOM UNIVERSITAS ESA UNGGUL	Pertemuan OL – IX

Materi

1. Algoritma Penyelesaian Mutex
2. Deadlock

MUTUAL EXCLUSION (MUTEX)

Kriteria Penyelesaian Mutual Exclusion

1. *Mutual exclusion* harus dijamin
Seksi kritis dijamin hanya diakses oleh satu proses pada satu saat
Hanya satu proses pada satu saat yang diijinkan masuk *critical section*. Proses – proses lain dilarang masuk *critical section* yang sama pada saat telah terdapat proses masuk di *critical section* itu.
2. Proses yang berada di *noncritical section*, dilarang mem – *blocked* proses – proses lain yang ingin masuk *critical section*.
3. Harus dijamin proses yang ingin masuk *critical section* tidak menunggu selama waktu yang tak berhingga. Atau tak boleh terdapat *deadlock* atau *starvation*.
4. Ketika tidak ada proses pada *critical section* maka proses yang ingin masuk *critical section* harus diijinkan masuk **tanpa waktu tunda**.
5. Tidak ada asumsi mengenai kecepatan relatif proses atau jumlah proses yang ada.

Metoda-Metoda Dasar Penyelesaian MUTEX antara lain :

1. Metoda Variabel Lock Sederhana

Menggunakan Variabel Lock sebagai pengunci Seksi Kritis

Jika Lock = 1 maka proses tidak boleh memasuki Seksi Kritis

Jika Lock = 0 maka proses dipersilahkan memasuki Seksi Kritis dan segera menset

Lock = 1 agar proses lain tidak memasuki Seksi Kritis.

Masalah : Proses paralel terjadi pada instruksi dasar, sehingga tidak dapat dipaksa suatu proses tidak memasuki seksi kritis ketika proses yang duluan masuk belum sempat menset Variabel Lock.

2. Metoda Pergantian Secara Ketat

Menggunakan Variabel Turn dengan nilai yang berbeda antara suatu proses dengan proses lainnya.

Masuknya proses-proses terjadi pada prosedur.

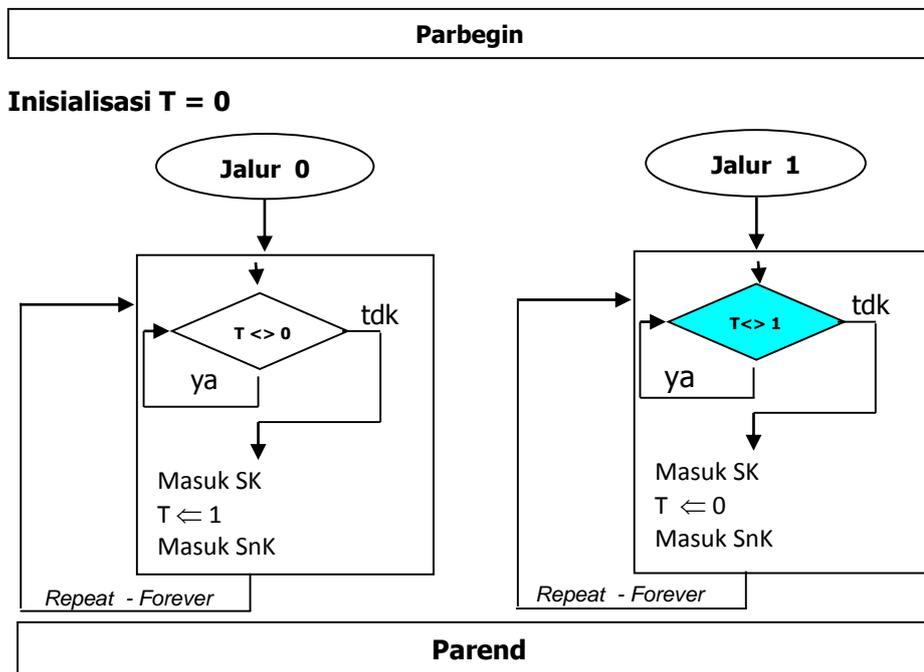
3. Algoritma Peterson

4. Semaphore

Semaphore adalah teknik penyelesaian Mutex dengan memanfaatkan Queue

Metoda Bergantian Secara Ketat

Metoda ini mengatur supaya urutan giliran antara beberapa proses menjadi lebih adil. Metoda mengizinkan parallel dua proses pada saat yang sama.



Data proses :

Id proses	waktu porses datang	ukuran paket data
P0	0	10
P1	1	2
P2	4	2
P3	20	3

Status Awal : turn = 0, nomor slot kosong berikutnya = 4

Digunakan sebuah variabel *turn* sebagai kendali seksi kritis :

- Jika variabel *turn* = 0 maka proses di jalur 0 diijinkan masuk seksi kritis, dan proses di jalur 1 blocked
- Jika variabel *turn* = 1 maka proses di jalur 1 diijinkan masuk seksi kritis dan proses di jalur 0 blocked

<pre> Program Mutex - with -strict -alternation; Var turn : Integer; Procedure enter -critical -section; { Mengerjakan kode - kode kritis } Procedere enter -noncritical -section; { Mengerjakan kode - kode tak kritis } Procedure Proses 0; Begin Repeat While turn <> 0 Do Begin End; enter -critical -section; turn := 1; enter -noncritical -section; Forever End; </pre>	<pre> Procedure Proses 1; Begin Repeat While turn <> 1 Do Begin End; enter -critical -section; turn := 0; enter -noncritical -section; Forever End; Begin turn := 0; Parbegin Proses 0; Proses 1; Parend End. </pre>
--	--

Kelemahannya → looping busy waiting hanya memeriksa nilai kunci, dan tidak memeriksa apakah proses yang menghalangi berada di SK atau bukan.

Metoda ini dapat menghindari *race condition* tapi gagal menyelesaikan Mutex karena proses yang berada di seksi non kritis dapat menghalangi proses lain masuk ke seksi kritis. Perhatikan contoh berikut ini.

Tabel Data Proses :

Id proses	waktu porses datang	ukuran paket data
P0	0	10
P1	1	2
P2	4	2
P3	20	3

Status Awal : turn = 0, nomor slot kosong berikutnya = 4

Clock	P0	P1	P2	P3	SK	Proses Dieksekusi	Deskripsi
0.	Jalur 0				Idle	P0	Turn <> 0 => 0<>0 = F; P0 diizinkan masuk SK
1.	In SK	Jalur 1			P0	P0	In SK => get slot P0= 4
2.	In SK	blocked			P0	P1	Turn <> 1 => 0<>1 = T; P1 blocked
3.	In SK	blocked			P0	P0	In SK => update slot = 5
4.	In SK	blocked	Delay		P0	P1	Turn <> 1 => 0<>1 = T; P1 blocked
5.	In SK	blocked	Delay		P0 out	P0	Turn <= 1
6.	In SNK	In SK	Delay		P1	P1	Turn <> 1 => 1<>1 = F; P1 diizinkan masuk SK
7.	In SNK	In SK	Delay		P1	P0	P0 In SNK (1)
8.	In SNK	In SK	Delay		P1	P1	In SK => get slot P1=5
9.	In SNK	In SK	Delay		P1	P0	P0 In SNK (2)
10.	In SNK	In SK	Delay		P1	P1	In SK => update Slot = 6
11.	In SNK	In SK	Delay		P1	P0	P0 In SNK (3)
12.	In SNK	In SK	Delay		P1 out	P1	Turn <= 0
13.	In SNK	inSNK	Delay		Idle	P0	P0 In SNK (4)
14.	In SNK	inSNK	Delay		Idle	P1	P1 In SNK (1)
15.	In SNK	inSNK	Delay		Idle	P0	P0 In SNK (5)
16.	In SNK	inSNK	Delay		Idle	P1	P1 In SNK (2) P1 Selesai jalur 1 kosong
17.	In SNK		Jalur 1		Idle	P0	P0 In SNK (6)
18.	In SNK		blocked		Idle	P2	Turn <> 1 → 0 <> 1 = True → P2 blocked padahal SK idle
19.	In SNK		Blocked		Idle	P0	P0 In SNK (7)
20.	In SNK		blocked	Delay	Idle	P2	Turn <> 1 → 0 <> 1 = True → P2 blocked padahal SK idle
21.	In SNK		blocked	Delay	Idle	P0	P0 In SNK (8)
22.	In SNK		blocked	Delay	Idle	P2	Turn <> 1 → 0 <> 1 = True → P2 blocked padahal SK idle
23.	In SNK		blocked	Delay	Idle	P0	P0 In SNK (9)
24.	In SNK		blocked	Delay	Idle	P2	Turn <> 1 → 0 <> 1 = True → P2 blocked padahal SK idle
25.	In SNK		blocked	Delay	Idle	P0	P0 In SNK (10) P0 Selesai jalur 0 kosong P3 masuk jalur 0
26.			blocked	Jalur 0	Idle	P3	
27.			blocked				
28.			blocked				

clock ke 16, P1 menyelesaikan prosesnya, sehingga jalur 1 kosong

clock ke 17, P2 mendapat alokasi jalur 1,

clock ke 18, P2 gagal masuk seksi kritis, padahal seksi kritis kosong

Metode Penyelesaian Peterson (2 proses) (perbaikan metoda Bergantian Secara Ketat)

Mekanisme kerja Alg. Peterson adalah sebagai berikut :

Terdapat dua variabel kendali seksi kritis :

- Array `interested[i]` dengan type Boolean \rightarrow true, false \rightarrow $i = 0$ dan 1
- Variabel bersama `Turn`

Proses 0

`Interested [0]` untuk mengidentifikasi status `proses_0`

Jika `Interested [0]` bernilai `true` berarti `proses_0` ada dan akan/sedang masuk_SK

Dan jika `Turn = 0` maka `proses_0` memasuki seksi kritis dan memblok `proses_1`
Tapi jika `Turn = 1` dan `proses_1` ada, maka `proses_0` akan diblok

Looping kendali : `While interested [1] and turn = 1 do begin end`

Jika `Interested [0]` bernilai `false` berarti `proses` di jalur 0 tidak ada atau sedang di SnK

Proses 1

`Interested [1]` untuk mengidentifikasi status `proses_1`

Jika `Interested [1]` bernilai `true` berarti `proses_1` ada akan/sedang masuk_SK

Dan jika `Turn = 1` maka `proses_1` memasuki seksi kritis dan memblok `proses_0`
Tapi jika `Turn = 0` dan `proses_0` ada, maka `proses_1` akan diblok

Looping kendali : `While interested [0] and turn = 0 do begin end`

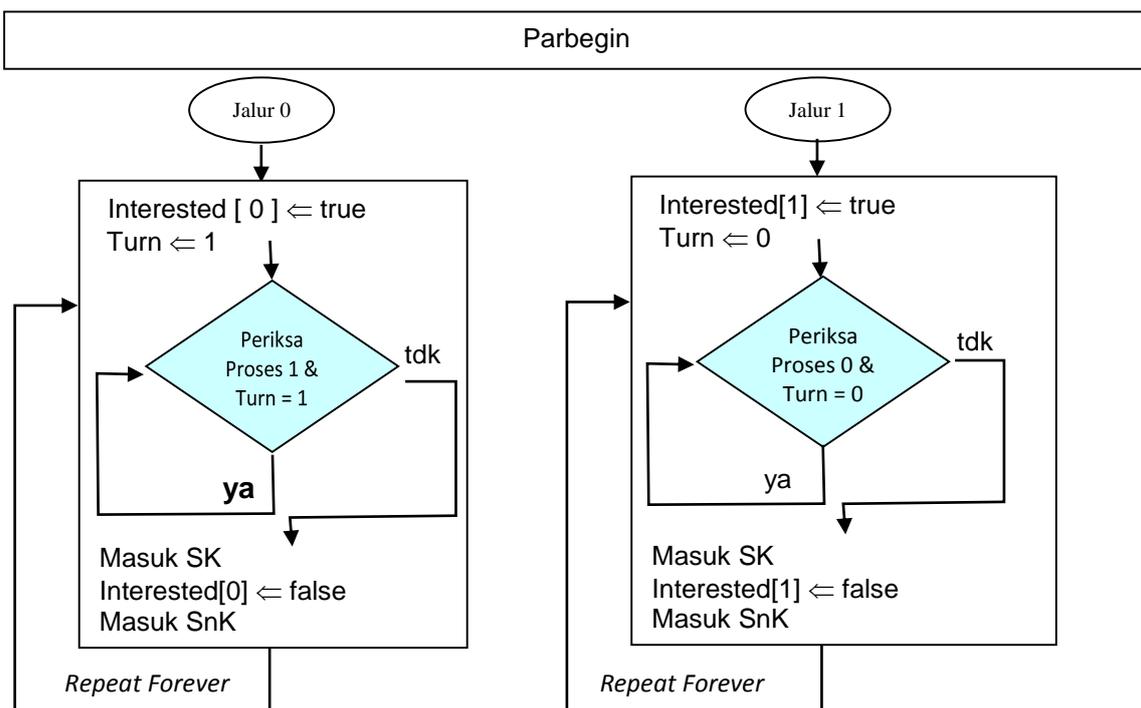
Note :

Ketika datang `proses_0`, `Interested [0]` akan diberi nilai `true`, dan `turn = 1`

Ketika datang `proses_1` `Interested [1]` akan diberi nilai `true`, dan `turn = 0`

Jika dua proses datang secara parallel, maka proses yang terakhir melaksanakan intruksi pengisian nilai `turn`, akan diblok.

Flowchart dari Algoritma Peterson



<p>1 Algoritma Peterson; 2 Const n = 2 3 Var 4 Interested : array [0..n-1] of Boolean 5 Turn : integer</p>	
<p>6 Prosedur Masuk_SK; { mengerjakan instruksi-instruksi di Seksi Kritis } 7 n <= vb vb <= n + 1 Prosedur Masuk_SnK; { mengerjakan instruksi-instruksi di Seksi non Kritis }</p>	
<p>8 Prosedur Proses_0; 9 Begin 10 Repeat 11 Interested[0] <= true 12 Turn <= 1 13 While interested [1] and turn = 1 do begin end 14 Masuk_SK; 15 Interested [0] <= false 16 Masuk_SnK 17 Forever 18 End-Proses_0</p>	
<p>19 Prosedur Proses_1; 20 Begin 21 Repeat 22 Interested[1] <= true 23 Turn <= 0 24 While interested [0] and turn = 0 do begin end 25 Masuk_SK; 26 Interested [1] <= false 27 Masuk_SnK 28 Forever 29 End-Proses_1</p>	
<p>30 Begin 31 Interested [0] <= false 32 Interested [1] <= false 33 Turn <= 1 34 Parbegin 35 Proses_0 36 Proses_1 37 Parend 38 End-Peterson</p>	

Algoritma penyelesaian Peterson dapat dideskripsikan sebagai berikut :

Data proses :

P0	0	10
P1	1	2
P2	4	2
P3	20	3

Turn = 0 nomor Slot berikutnya → 3

Clock	Action		P0	P1	P2	P3
0.	Int[0] ← True	P0	Jalur 0			
1.	Int[1] ≤ true	P1		Jalur 1		
2.	Turn = 1	P0				
3.	Turn = 0	P1			delay	
4.	Int[1] ∧ Turn = 1 : T ∧ T = T → P0 blocked	P0	Blocked		delay	
5.	Int[0] ∧ Turn = 0 : T ∧ F = F → P1 PASS	P1	Blocked	Pass	delay	
6.	Int[1] ∧ Turn = 1 : T ∧ T = T → P0 blocked	P0	Blocked	Pass	delay	
7.	P1 In SK get slot 3; slot => 4	P1	Blocked	Pass	delay	
8.	Int[1] ∧ Turn = 1 : T ∧ T = T → P0 blocked	P0	Blocked		delay	
9.	Int[1] ≤ F	P1	Blocked	Out SK	delay	
10.	Int[1] ∧ Turn = 1 : F ∧ T = F → P0 Pass	P0	Pass		delay	
11.	P1 in SNK (1)	P1			delay	
12.	P0 in SK get slot 4; slot => 5	P0			delay	
13.	P1 in SNK (2) selesai	P1		Finish	Jalur 1	
14.	Int[0] ≤ false	P0				
15.	Int[1] ≤ true	P2				
16.	P0 in SnK (1)	P0				
17.	Turn = 0	P2				
18.	P0 in SnK (2)	P0				
19.	Int[0] ∧ Turn = 0 : F ∧ T = F → P2 PASS	P2				
20.	P0 in SnK (3)	P0				DELAY
21.	P2 in SK get slot 5; slot => 6	P2				DELAY
22.	P0 in SnK (4)	P0				DELAY
23.	Int[1] ≤ false	P2				DELAY
24.	P0 in SnK (5)	P0				DELAY
25.	P2 in SnK (1) selesai	P2				Jalur 1
26.	P0 in SnK (6)	P0				
27.	Int [1] ≤ true	P3				
28.	P0 in SNK (7)	P0				Jalur 1
29.	Turn = 0	P3				
30.	P0 in SNK (8)	P0				
31.	Int[0] ∧ Turn = 0 : F ∧ T = F → P3 PASS	P3				P3 pass
32.		P0				P3 pass
33.		P2				P3 pass
34.		P3				P3 pass
35.		P2				P3 pass
36.		P3				P3 pass
37.		P2				P3 pass

SEMAPHORE

Deskripsi Semaphore

Semaphore adalah pendekatan yang dikemukakan Dijkstra.

Prinsip semaphore sebagai berikut :

- Dua proses atau lebih dapat bekerja sama dengan menggunakan penanda – penanda sederhana.
- Proses dipaksa berhenti sampai proses memperoleh penanda tertentu.
- Sembarang kebutuhan koordinasi kompleks dapat dipenuhi dengan struktur penanda yang sesuai kebutuhannya.
- **Variabel khusus untuk penandaan ini disebut semaphore.**

Terdapat dua operasi terhadap semaphore yaitu Down dan Up → *atomik* (sebuah procedure yang harus dieksekusi dalam 1 paket waktu → tidak dapat di-preempt)
Semaphore diimplementasikan dengan dukungan sebuah *Queue*

Operasi Down

- Operasi ini menurunkan nilai semaphore.
- Jika nilai semaphore menjadi nonpositif maka proses di-blocked.

```
Procedure Down ( Var s : semaphore );  
Begin  
  s := s - 1 ;  
  If s < 0 Then  
    Begin  
      Tempatkan proses pada antrian untuk semaphore s  
      Proses di – blocked  
    End;  
End;
```

Operasi Up

Operasi Up menaikkan nilai semaphore, jika nilai semaphore nonpositif atau 0 pindahkan satu proses dari antrian dan menempatkan satu proses ke senarai ready

Semaphore non positif atau 0 → menunjukkan antrian tidak kosong

```
Procedure Up ( Var s : semaphore )  
Begin  
  s := s + 1 ;  
  If s <= 0 Then  
    Begin  
      Pindahan satu proses P dari antrian untuk semaphore s  
      Tempatkan proses P di senarai ready  
    End;  
End;
```

Semaphore →

sebuah variabel sederhana yang nilai-nilai nya mengidentifikasi keadaan:

- status SK → kosong atau isi
- status Queue → ada atau tidaknya proses didalam antrian

MUTUAL EXCLUSION DENGAN SEMAPHORE

Adanya semaphore mempermudah persoalan mutual exclusion.

Skema penyelesaian mutual exclusion mempunyai bagan program sebagai berikut:

```

Program Mutual - exclusion - with - Semaphore;
Const
  N = ?;
Var
  s : semaphore;

Procedure enter - critical - section;
{ Mengerjakan kode – kode kritis}

Procedure enter - noncritical - section;
{ Mengerjakan kode – kode non kritis }

Procedure Proses ( i : integer );
Begin
  Down ( s );
  enter - critical - section;
  Up ( s );
  enter - noncritical - section;
End;

Begin
  s := 1 ;    { init atau nilai awal semaphore}
  i := 0
  Repeat
    Parbegin
      Proses ( i );
      i = ( i + 1 ) mod N
    Parend
  Forever
End.

```

```

Procedure Down ( Var s : semaphore );
Begin
  s := s - 1 ;
  If s < 0 Then
    Begin
      Tempatkan proses pada antrian untuk
      semaphore s. Proses di - blocked,
      menunggu diaktifkan oleh proses lain
      melalui prosedur UP
    End-if;
  End

Procedure Up ( Var s : 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-if
  End

```

Contoh

Kasus Printer Daemon, SK adalah :

```

Procedure enter - critical - section;
{ Mengerjakan kode – kode kritis}
Begin
  dapat_slot := next_slot
  next_slot := dapat_slot +1
end

```

Data proses :

Id proses	Waktu datang	Jumlah siklus waktu store berkas
P0	0	2
P1	1	2
P2	4	2
P3	20	3

S = 1 nomor Slot berikutnya → 3

Clock	Action		P0	P1	P2	P3
0.	Down(s) S = 0 → P0 Pass	P0	datang			
1.	Down(s) S = -1 → P1 masuk Queue	P1	Down	Datang		
2.	P0 in SK Dapat slot 3	P0	In SK	in queue		
3.	Next Slot = 4	P0	In SK	In queue		
4.	Down(s) S = -2 → P2 masuk Queue	P2	Selesai SK	In queue	Datang	
5.	Up(s) S = -1 → Q ≠ nihil → Bebaskan satu proses → P1	P0	Up	Keluar dari Queue	In queue	
6.	P1 in SK Dapat slot 4	P1	In SNK	In SK	In queue	
7.	P0 in SNK Store (1)	P0	In SNK	In SK	In queue	
8.	Next slot = 5	P1	In SNK	In SK	In Queue	
9.	P0 in SNK Store (2) dan selesai	P0	In SNK selesai	Selesai SK		
10.	Up (s) S = 0 → Q ≠ nihil → Bebaskan satu proses → P2	P1		Up	Keluar dari Queue	
11.	P2 in SK Dapat slot 5	P2		In SNK		
12.	P1 in SNK Store (1)	P1		In SNK		
13.	Next slot = 6	P2		In SNK		
14.	P1 in SNK Store (2) dan selesai	P1		In SNK selesai		
15.	Up(s) S = 1 → Q = nihil	P2				
16.	P2 in SNK Store (1)	P2				
17.	P1 in SNK Store (2) dan selesai	P2				
18.	Idle					
19.	Idle					
20.	P3	P3				
21.						
22.						

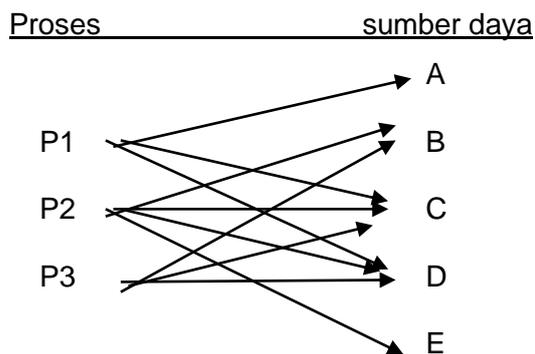
- Sebelum masuk critical section, proses melakukan Down
→ output dari down : pass atau terblock.
 - Bila berhasil maka proses masuk critical section .
 - Bila tidak berhasil maka proses di – blocked pada semaphore itu.
- Proses yang di – blocked akan dapat melanjutkan kembali bila proses di critical section keluar dan melakukan operasi Up sehingga menjadikan proses yang di – blocked ready dan berlanjut sehingga operasi Down – nya berhasil.

DEADLOCK

Deadlock adalah suatu kondisi dimana 2 proses atau lebih tidak dapat meneruskan eksekusinya. Proses-proses disebut deadlock jika proses-proses tersebut saling menunggu suatu kejadian yang tidak akan pernah terjadi.

Deadlock terjadi ketika proses-proses mengakses secara eksklusif sumber daya. Semua deadlock yang terjadi melibatkan persaingan proses-proses memperoleh sumber daya eksklusif oleh dua proses atau lebih.

Ilustrasi terjadinya deadlock :



Skenario terjadinya Deadlock

- P1 P2 dan P3 datang bersamaan.
- P1 mendapat alokasi A,
- P2 mendapat alokasi B
- P3 mendapat alokasi C

Fase berikutnya

- P1 tetap menggenggam A dan mendapatkan D,
- P2 tetap menggenggam B dan mendapatkan E
- P3 tetap menggenggam C dan menunggu B atau D, sementara P3 tidak dapat meneruskan prosesnya
- P1 tetap menggenggam A, D dan menunggu C, sementara P1 tidak dapat meneruskan prosesnya
- P2 tetap menggenggam B, E dan menunggu C atau D, sementara P2 tidak dapat meneruskan prosesnya
- Semua proses tidak dapat terus karena sumber daya yang ditunggu sedang digenggam oleh proses lain yang tidak dapat melepaskannya sampai prosesnya selesai.

Secara umum kejadian ini dapat mudah terjadi dalam pemrograman multi-thread. Sebab ada kemungkinan lebih besar untuk menggunakan sumber daya bersama.

Sumber Daya yang Bisa Dipakai Berulang-Ulang

Kejadian *deadlock* selalu tidak lepas dari sumber daya, seperti kita lihat dari contoh contoh diatas, bahwa hampir seluruhnya merupakan masalah sumber daya yang digunakan bersama-sama. Oleh karena itu, kita juga perlu tahu tentang jenis sumber daya, yaitu: sumber daya dapat digunakan lagi berulang-ulang dan sumber daya yang dapat digunakan dan habis dipakai atau dapat dikatakan sumber daya sekali pakai.

Sumber daya ini tidak habis dipakai oleh proses manapun. Tetapi setelah proses berakhir, sumber daya ini dikembalikan untuk dipakai oleh proses lain yang sebelumnya tidak kebagian sumber daya ini.

Contohnya prosesor, kanal I/O, disk, semaphores. Contoh peran sumber daya jenis ini pada terjadinya *deadlock* ialah misalnya sebuah proses memakai disk A dan B, maka akan terjadi *deadlock* jika setiap proses sudah memiliki salah satu disk dan meminta disk yang lain.

Masalah ini tidak hanya dirasakan oleh pemrogram tetapi oleh seorang yang merancang sebuah sistem operasi. Cara yang digunakan pada umumnya dengan cara memperhitungkan dahulu sumber daya yang digunakan oleh proses-proses yang akan menggunakan sumber daya tersebut.

Contoh lain yang menyebabkan *deadlock* dari sumber yang dapat dipakai berulang-ulang ialah berkaitan dengan jumlah proses yang memakai memori utama

Contohnya dapat dilihat dari kode berikut ini:

```
//dari kelas proses kita tambahkan method yaitu meminta
public void meminta (int banyakA) {
    //meminta dari sumber daya a
    if ( banyakA < banyak )
        banyak = banyak - banyakA;
    else
        wait();
}

//mengubah kode pada mainnya sebagai berikut
public static void main ( String [] args ) {
    Proses P = new Proses();
    Proses Q = new Proses();
    P.meminta(80);
    Q.meminta(70);
    P.meminta(60);
    Q.meminta(80);
}

private int banyak = 200;
private int banyakA;
```

Setelah proses P dan Q telah melakukan fungsi meminta untuk pertama kali, maka sumber daya yang tersedia dalam banyak ialah 50 (200-70- 80). Maka saat P menjalankan fungsi meminta lagi sebanyak 60, maka P tidak akan menemukan sumber daya dari banyak sebanyak 60, maka P akan menunggu hingga sumber daya yang diminta dipenuhi. Demikian juga dengan Q, akan menunggu hingga permintaannya dipenuhi, akhirnya terjadi *deadlock*.

Cara mengatasinya dengan menggunakan memori maya.

Sumber Daya Sekali Pakai

Dalam kondisi biasa tidak ada batasan untuk memakai sumber daya apapun, selain itu dengan tidak terbatasnya produksi akan membuat banyak sumber daya yang tersedia. Tetapi dalam kondisi ini juga dapat terjadi *deadlock*.

Contohnya dapat dilihat dari kode berikut ini:

```
//menambahkan method receive dan send
public void receive( Proses p ){
    //method untuk menerima sumber daya
}
public void send ( Proses p ){
    //method untuk memberi sumber daya
}
```

dari kedua fungsi tersebut ada yang bertindak untuk menerima dan memberi sumber daya, tetapi ada kalanya proses tidak mendapat sumber daya yang dibuat sehingga terjadi blok, karena itu terjadi *deadlock*. Tentu saja hal ini sangat jarang terjadi mengingat tidak ada batasan untuk memproduksi dan mengkonsumsi, tetapi ada suatu keadaan seperti ini yang mengakibatkan *deadlock*. Hal ini mengakibatkan *deadlock* jenis ini sulit untuk dideteksi. Selain itu *deadlock* ini dihasilkan oleh beberapa kombinasi yang sangat jarang terjadi.

KARAKTERISTIK DEADLOCK

Deadlock terjadi jika sifat-sifat berikut terpenuhi :

1. Mutual Exclusion
Jika suatu proses telah menggunakan suatu sumber daya, maka proses lain tidak diizinkan menggunakan sumber daya tersebut.
2. Hold and wait
Suatu proses tidak dapat melepaskan sumber daya yang sudah digenggamnya hingga proses tersebut mendapatkan seluruh sumber daya yang dibutuhkan dan menyelesaikan prosesnya.
3. No Preemption
Jika proses sudah mendapatkan suatu sumber daya, maka tidak ada proses lain yang diizinkan menyelanya.
4. Circular Wait
Jika proses-proses berada dalam lingkaran saling menunggu.

Deadlock benar-benar terjadi jika ke empat syarat tersebut terpenuhi. 3 syarat yang pertama disebut sebagai syarat perlu yang merepresentasikan sifat sistem, dan syarat ke empat disebut sebagai syarat harus yang merepresentasikan kejadian suatu keadaan.

METODA MENGATASI DEAD LOCK

Terdapat beberapa metoda mengatasi deadlock, secara umum dikelompokkan dalam 3 metoda yaitu :

- Metode Deadlock Prevention
- Metode Deadlock Avoidence
- Metode Deadlock Detection & Recovery

Strategi burung Onta

Strategi ini mengasumsikan kejadian deadlock jarang terjadi, sehingga mengabaikan kemungkinan deadlock. Jika terjadi deadlock maka *reboot* sistem. Strategi ini berarti sama sekali tidak berusaha mengatasi deadlock.

Pencegahan Deadlock (Deadlock Prevention)

Menghilangkan salah satu syarat perlu terjadinya deadlock. Jika salah satu syarat dari keempat syarat deadlock tidak terpenuhi maka tidak akan pernah terjadi deadlock.

Pencegahan dealock dapat dilakukan dengan :

1. meniadakan mutex,

deadlock terjadi karena adanya tuntutan penggunaan eksklusif sumber daya. Jika sumber daya selalu dapat dishare, maka tidak akan pernah terjadi deadlock

2. meniadakan syarat hold & wait, atau

untuk menghindari kondisi hold-wait dapat dilakukan upaya :

- mengizinkan alokasi jika semua sumber daya yang diperlukan tersedia.
Jika terdapat sumber daya yang belum tersedia maka tidak dilakukan alokasi sama sekali
- hold and release
jika suatu proses membutuhkan sumberdaya lainnya, maka proses tersebut mesti melepaskan sumber daya yang sudah digenggamnya

3. meniadakan kondisi no-preemption

Misalnya : sistem menetapkan tidak semua proses boleh disela (preemption), maka jika terjadi kondisi saling menunggu, maka salah satu proses dapat diinterrupt dan dipaksa melepaskan sumber daya yang digenggamnya.

4. menghindari kondisi menunggu secara sirkular

Untuk menghindari kondisi menunggu secara sirkular, maka dapat dilakukan dengan cara berikut :

- proses hanya boleh menggenggam satu sumber daya pada satu saat
- memberi nomor urut global terhadap semua permintaan sumber daya

Penghidaran Deadlock (Deadlock Avoidance)

Memberikan akses kepada sumber daya yang menjamin tidak terjadi deadlock. Dengan strategi ini, selalu dihindarkan pengalokasian sumber daya yang dapat beresiko deadlock.

→ alokasi sumber daya secara terkendali
menghindarkan terjadinya circular wait

Teknik yang ditawarkan :

- State Selamat dan State Tak Selamat
- Algoritma Banker

State Selamat dan State Tak Selamat

- State selamat : jika terdapat cara untuk memenuhi semua permintaan yang tertunda tanpa menimbulkan deadlock dengan menjalankan proses secara hati-hati mengikuti suatu aturan tertentu
- State tak selamat : Jika tidak terdapat cara untuk memenuhi semua permintaan yang tertunda tanpa menimbulkan deadlock

Algoritma Banker

- Mengijinkan kondisi mutex, hold and wait dan no-preemption
- Alokasi sumber daya hanya dilakukan jika menghasilkan state selamat

Contoh

Sumber daya : 30

- dapat diperebutkan secara bebas = 15
- alokasi terkendali = 15

State Selamat

Id proses	Need	Alokasi awal	sisa	I		II		III			
				alokasi	Sisa	alokasi	sisa	Alokasi	sisa		
P1	10	4	6	6	P1 completion dan membebaskan 10						
P2	12	4	8	3	5	2	3	3	Semua proses completion		
P3	15	4	11	3	8	3	5	5			
P4	11	3	8	3	5	5	P4 selesai bebaskan 11				

- alokasi terkendali = 15

State tidak selamat

Id proses	Need	Alokasi awal	sisa	I		II		III			
				alokasi	Sisa	alokasi	sisa	Alokasi	sisa		
P1	10	4	6	4	2						
P2	12	4	8	4	4						
P3	15	4	11	4	7						
P4	11	3	8	3	5						

Contoh :

1. Sebuah sistem memiliki 10 sumber daya, 5 diperebutkan secara bebas dan 5 lainnya dialokasikan secara terkendali, terdapat 3 proses yang mengakses sistem secara bersamaan, yaitu :
 - Proses A memerlukan sumber daya maksimum 10,
 - Proses B memerlukan sumber daya maksimum 4
 - Proses C memerlukan sumber daya maksimum 8
 - a. Susunlah skenario Genggam dan Tunggu yang menyebabkan sistem tersebut terhindar dari deadlock (state selamat/ safe state)
 - b. Susunlah skenario Genggam dan Tunggu yang menyebabkan sistem tersebut deadlock (state tdk selamat/ unsafe state)

State Tidak Selamat

Id proses	Need	Alokasi awal	sisa	I		II		III			
				alokasi	Sisa	alokasi	sisa	Alokasi	sisa		
PA	10	1	9	2	7	0					
PB	4	2	2	2	PB selesai dan membebaskan 4 sumber daya						
PC	8	2	6	1	5	4					

Akibat pada tahap alokasi terkendali pertama : alokasi 2 untuk Pa dan 1 Pc → tidak ditemukan cara alokasi berikutnya yang dapat menuju keadaan sukses → *state tidak selamat*

Id proses	Need	Alokasi awal	sisa	I		II		III			
				Alokasi	Sisa	alokasi	sisa	Alokasi	sisa		
PA	10	1	9	1	8	0	8	8		selesai	
PB	4	2	2	2	PB selesai dan membebaskan 4 sumber daya						
PC	8	2	6	2	4	4	PC selesai				

Id proses	Need	Alokasi awal	sisa	I		II		III			
				Alokasi	Sisa	alokasi	sisa	Alokasi	sisa		
PA	10	2	9	0	8	0	8	8		selesai	
PB	4	2	2	2	PB selesai dan membebaskan 4 sumber daya						
PC	8	1	6	3	4	4	PC selesai				

ALGORITMA BANKER

Implementasi dalam perbankan state selamat state tidak selamat → algoritma Banker
→ alokasi kredit (pembiayaan)

2 T, 20 proposal @200M → diasumsikan kebutuhan valid

Penyelesaian :

2 T : 1 T alokasi rata, 1 T alokasi terkendali

- a. setiap proposal mendapat investasi awal @50M sisa 150M
- b. analisa proyek yang ada, pilih maks 6 proposal untuk mendapat alokasi penuh :
proyek yang dapat segera bayar utang
misal 5 proyek → 150 M → 750 M sisa 250M
sisa 250 M alokasi rata :
 - o untuk 14 proyek lain @ 16M → 50+16 = 66 M sisa 134M
 - o 1 proyek dapat 26 M → 76 M sisa 124M
- c. Pengembalian dari 5 proyek tercepat : 1 T
analisa proyek yang ada, pilih maks 7 proposal berikutnya untuk mendapat alokasi penuh : proyek yang dapat segera bayar utang
misal 1 proyek → 124 M → 124 M
5 proyek → 134 M → 670 M
sisa 206 M alokasi rata :
 - o untuk 8 proyek lain @ 20 M → 86 M sisa 114M
 - o 1 proyek dapat 46 M → 112M sisa 88M
- d. Pengembalian dari 6 proyek tercepat : 1,2 T
untuk 8 proyek lain punya 86 M + 114M dapat dipenuhi semua
1 proyek sudah punya 112M + 88M dapat dipenuhi
→ DANA SISA 200 M

Deteksi dan Pemulihan Deadlock (Deadlock Detection & Recovery)

Membiarkan deadlock terjadi, jika kemudian terjadi lakukan upaya penyelesaian.
Terdiri dari dua langkah pokok : eksi adanya deadlock

- Pemulihan Deadlock

Jika diketahui terjadi deadlock, maka deadlock harus diputuskan dengan menghilangkan salah satu atau lebih sarat perlu terjadinya deadlock.

Hal-hal yang merumitkan pemulihan deadlock :

1. Belum dapat memutuskan apakah deadlock benar terjadi
2. Tidak dimiliki mekanisme suspend proses yang memadai.
3. Diperlukan kemampuan operator yang memadai untuk suspend secara efektif

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.

Ada beberapa jalan untuk pemulihan *deadlock*:

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 untuk 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.

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. Meskipun 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.

Membunuh proses yang menyebabkan deadlock

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