

# **Sinkronisasi dan *Deadlock***

# Latar Belakang Sinkronisasi (1)

- ❖ Mengapa perlu dilakukan sinkronisasi?
  - Sinkronisasi diperlukan untuk menghindari terjadinya ketidakkonsistenan data akibat adanya akses data secara konkuren
  - Untuk menjaga kekonsistenan data tersebut, diperlukan adanya suatu mekanisme untuk memastikan urutan pengaksesan suatu data yang saling bekerjasama sehingga terjadi sinkronisasi

# Latar Belakang Sinkronisasi (2)

- ❖ Masalah-masalah yang dapat timbul bila sinkronisasi tidak diterapkan
  - Masalah *Bounded-Buffer*
  - *Race Condition*

# *Bounded-Buffer (1)*

❖ Proses yang dilakukan oleh produsen:

```
item nextProduced;
while (1)
{
    while (counter == BUFFER_SIZE) { ... do
nothing ... }
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

# *Bounded-Buffer (2)*

❖ Proses yang dilakukan oleh konsumen:

```
item nextConsumed;  
while (1)  
{  
    while (counter == 0) { ... do nothing ... }  
    nextConsumed = buffer[out] ;  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
}
```

# *Bounded-Buffer (3)*

- ❖ Perhatikan baris-baris yang memuat perintah di bawah ini:
  - `counter++;`
  - `counter--;`
- ❖ Perintah-perintah tersebut adalah proses yang harus diselesaikan secara langsung tanpa adanya interupsi dari proses lain; proses yang memiliki karakteristik seperti itu juga disebut proses yang harus dijalankan secara atomik

# *Bounded-Buffer (4)*

- ❖ Perintah “**count++**” bisa diimplementasikan pada bahasa mesin menjadi:

**register1 = counter**

**register1 = register1 + 1**

**counter = register1**

- ❖ Perintah “**count--**” bisa diimplementasikan pada bahasa mesin menjadi:

**register2 = counter**

**register2 = register2 - 1**

**counter = register2**

# ***Bounded-Buffer (5)***

- ❖ Jika kedua perintah tersebut berusaha mengakses nilai **counter** secara konkuren, maka dapat terjadi kesalahan pada nilai **counter** karena sifat bahasa mesin yang menggunakan register untuk mengupdate nilai **counter**
- ❖ Kesalahan nilai akhir **counter** dapat terjadi, tergantung dari penjadwalan yang dilakukan terhadap proses yang dilakukan oleh produsen dan konsumen. Dengan kata lain, masalah tersebut belum tentu terjadi, tapi dapat terjadi

# ***Bounded-Buffer (6)***

- ❖ Misalnya nilai **counter** adalah 2. Bila dilakukan proses produsen dan konsumen secara konkuren:

produsen: **register1 = counter** (*register1 = 2*)

produsen: **register1 = register1 + 1** (*register1 = 3*)

konsumen: **register2 = counter** (*register2 = 2*)

konsumen: **register2 = register2 - 1** (*register2 = 1*)

konsumen: **counter = register2** (*counter = 1*)

produsen: **counter = register1** (*counter = 3*)

## ***Bounded-Buffer (6)***

- ❖ Nilai akhir **counter** menjadi 3, padahal seharusnya tetap 2 setelah dilakukan sebuah proses oleh masing-masing produsen dan konsumen.
- ❖ Situasi ini terjadi karena program mengizinkan pengaksesan terhadap nilai **counter** secara konkuren, sehingga terjadilah suatu situasi yang biasa disebut *race condition*

# *Race Condition*

- ❖ *Race condition*: situasi dimana beberapa proses mengakses dan memanipulasi suatu data secara konkuren. Nilai akhir dari data tersebut tergantung dari proses mana yang terakhir selesai dieksekusi
- ❖ Untuk menghindari terjadinya situasi tersebut, semua proses yang dapat mengakses suatu data tertentu harus disinkronisasi

# Problema *Critical Section*

- ❖ Lebih dari satu proses berlomba-lomba pada saat yang sama untuk menggunakan data yang sama.
- ❖ Setiap proses memiliki segmen kode yang mengakses data yang digunakan secara bersama-sama. Segmen kode tersebut disebut *critical section*.
- ❖ **Masalahnya:** menjamin bahwa jika suatu proses sedang menjalankan *critical section*, maka proses lain tidak boleh masuk ke dalam *critical section* tersebut.

# Solusi dari Masalah *Critical Section* (1)

❖ Solusi dari problema *critical section* harus memenuhi tiga syarat berikut:

## 1. *Mutual Exclusion*

Tidak ada dua proses yang berada di *critical section* pada saat yang bersamaan.

## 2. **Terjadi kemajuan**

Jika tidak ada proses yang sedang berada di *critical section*, maka proses lain yang ingin menjalankan *critical section* dapat masuk ke dalam *critical section* tersebut.

# Solusi dari Masalah *Critical Section* (2)

## 3. Ada batas waktu tunggu

Tidak ada proses yang menunggu selama-lamanya untuk masuk ke dalam *critical section*.

- ❖ Diasumsikan bahwa setiap proses berjalan pada kecepatan yang bukan nol. Tidak ada asumsi lain mengenai kecepatan relatif setiap proses ataupun jumlah CPU.

# Dua Jenis Solusi Masalah *Critical Section*

- ❖ Solusi perangkat lunak

- Dengan menggunakan algoritma-algoritma yang nilai kebenarannya tidak tergantung pada asumsi-asumsi lain, selain bahwa setiap proses berjalan pada kecepatan yang bukan nol.

- ❖ Solusi perangkat keras

- Tergantung pada beberapa instruksi mesin tertentu, misalnya dengan me-non-aktifkan interupsi atau dengan mengunci suatu variabel tertentu.

# Langkah Awal untuk Memecahkan Masalah

- ❖ Hanya ada dua proses, yaitu  $P_0$  dan  $P_1$ .
- ❖ Struktur umum dari proses  $P_i$  (proses yang lain:  $P_j$ )

**do** {

*entry section*

*critical section*

*exit section*

*remainder section*

**} while (1);**

- ❖ Proses-proses tersebut boleh berbagi beberapa variabel yang sama untuk mensinkronisasikan apa yang akan dilakukan oleh setiap proses tersebut.

# Algoritma 1

❖ Variabel yang digunakan bersama:

→ `int turn;`

pada awalnya `turn = 0`

→ `turn = i;`  $\Rightarrow P_i$  dapat masuk ke *critical section*

❖ Proses  $P_i$

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

❖ Memenuhi syarat *mutual exclusion*, tetapi tidak memenuhi syarat terjadinya kemajuan.

# Algoritma 2

❖ Variabel yang digunakan bersama:

→ `boolean flag[2];`

pada awalnya `flag [0] = flag [1] = false.`

→ `flag [i] = true;`  $P_i$  dapat masuk ke *critical section*

❖ Proses  $P_i$

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

❖ Memenuhi syarat *mutual exclusion*, tetapi tidak memenuhi syarat terjadinya kemajuan.

# Algoritma 3

- ❖ Menggabungkan variabel yang digunakan bersama di algoritma 1 dan 2, dikenal juga dengan nama algoritma Peterson.
- ❖ Proses  $P_i$ 

```
do {  
    flag [i] := true;  
    turn = j;  
    while (flag [j] and turn = j);  
        critical section  
    flag [i] = false;  
        remainder section  
} while (1);
```
- ❖ Memenuhi tiga syarat untuk menyelesaikan masalah *critical section* pada dua proses.

# Algoritma Tukang Roti (1)

- ❖ Diperkenalkan pertama kali oleh Leslie Lamport.
- ❖ *Critical section* untuk  $n$  buah proses.
  - Sebelum memasuki *critical section*, setiap proses menerima sebuah nomor. Proses yang memegang nomor terkecil dapat masuk ke dalam *critical section*.
  - Jika proses  $P_i$  dan  $P_j$  menerima nomor yang sama, jika  $i < j$ , maka  $P_i$  dilayani dahulu, dan sebaliknya jika  $i > j$ , maka  $P_j$  dilayani dahulu. Dengan kata lain, yang memegang *ID* terkecil yang dilayani dahulu.
  - Skema penomoran selalu naik secara berurut, contoh: 1, 2, 2, 2, 3, 3, 4, 5...

# Algoritma Tukang Roti (2)

→ Notasi  $<$   $\equiv$  urutan leksikografikal (no tiket, no *ID* proses).

- $(a,b) < (c,d)$  jika  $a < c$  atau jika  $a = c$  dan  $b < d$
- $\max(a_0, \dots, a_{n-1})$  adalah sebuah bilangan  $k$ , sedemikian sehingga  $k \geq q_i$  untuk setiap  $i = 0, \dots, n - 1$

→ Data yang digunakan bersama

```
boolean choosing [n];
```

```
int number [n];
```

Struktur data diinisialisasi awal ke **false** dan **0**.

# Algoritma Tukang Roti (3)

```
do {
    choosing[I] = true;
    number[i] = max(number[0], number[1], ..., number[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]));
    }
    critical section
    number[i] = 0;
    remainder section
} while (1);
```

# Peran *Hardware* dalam Proses Sinkronisasi (1)

- ❖ Metode dalam sinkronisasi *hardware*
  - *Processor Synchronous* ( Interupsi )
  - *Memory Synchronous* ( Instruksi Test-And-Set )
- ❖ *Processor synchronous*
  - Dengan interupsi
  - Dalam lingkungan *multiprocessor* : Hanya satu processor yang tidak dapat diinterupsi
- ❖ *Memory synchronous*
  - Instruksi Test-And-Set
  - Dalam lingkungan *multiprocessor* : semua processor tidak dapat memakai resource karena proteksi dilakukan di memory
  - Instruksi harus bersifat atomik

# Peran *Hardware* dalam Proses Sinkronisasi (2)

- ❖ Contoh program *processor synchronization* ( dengan Atmel AVR™)

mainModul :

```
CLI          ' Masuk Critical Section dengan disable  
            ' interrupt  
ADD r1,r2   ' Critical Section  
.....     ' Critical Section  
SBI        ' Keluar dari Critical Section dengan enable  
            ' interrupt  
.....     ' Remainder Section
```

# Peran Hardware dalam Proses Sinkronisasi ( 3 )

- ❖ Contoh program *Memory synchronization* ( dengan Java™ )

```
boolean cekAndSet( boolean variable[] )
{
    boolean t = variable[0];
    variable[0] = true;
    return t;
}
....
while (cekAndSet(lock)) { ... do nothing ... }
// critical section
lock[0] = false;
// Remainder Section
```

# Instruksi Atomik

- ❖ Pengertian
- ❖ Perbedaan instruksi atomik dengan instruksi biasa
- ❖ Contoh instruksi atomic :
  - Intel Pentium™ : LOCK-Assert ( semua instruksi dapat dibuat atomik )
  - SPARC : swap, compare&swap
  - IBM 370 : CompareAndSwap
- ❖ TestAndSet harus bersifat atomik

# Sejarah Semafor

- ❖ Perkembangan penyelesaian *Critical Section* :
  - Masa sebelum 1960: *processor Synchronization*
  - 1967 : Konsep Semafor diajukan oleh **Dijkstra**
  - Masa sesudah 1960: Semafor banyak dipakai sebagai primitif

# Konsep Semafor (1)

❖ Pengertian

❖ Konsep

❖ Wait

→ *spinlock* (sibuk menunggu)

```
void waitSpinLock(int Semafor[] )
{
    while(Semafor[0] <= 0) { .. Do nothing ..
} //spinlock
    Semafor[0]--;
}
```

# Konsep Semafor ( 2)

→ tidak sibuk menunggu

```
void synchronized waitNonSpinLock(  
int semafor [])  
{  
    while(semafor[0] <= 0  
    {  
        wait(); // blocks thread  
    }  
    semafor[0] - -;  
}
```

# Konsep Semafor (3)

## ❖ Signal

→ *Untuk spinlock (sibuk menunggu)*

```
void signalSpinLock( int  semafor [])  
{  
    semafor[0]++;  
}
```

→ *Untuk tidak sibuk menunggu*

```
void synchronized signalNonSpinLock(int semafor[])  
{  
    semafor[0]++;  
    notifyAll();  
    // brings the waiting thread into ready queue  
}
```

**Note : instruksi wait dan signal harus bersifat atomik**

# Macam – macam Semafor

- ❖ Binary Semafor

- Nilai hanya berkisar 0 dan 1

- ❖ Counting Semafor

- Nilai tidak terikat pada 1 dan 0

# Semafor Menjawab Masalah Sinkronisasi !!

- ❖ Masalah Perjamuan Filsuf ( *Dining philosopher problem* )
- ❖ Masalah Pembaca dan Penulis ( *Readers-Writers problem* )
- ❖ Masalah Produsen dan Konsumen ( *Producer-Consumer problem* )

# Semafor di Sinkronisasi Tingkat Tinggi

- ❖ Implementasi *Critical Section* dengan Semafor
- ❖ Implementasi *Counting* Semafor dari *Binary* Semafor

# Semafor di dalam Pemrograman (1)

## ❖ Microsoft Windows™ Programming

→ Nilai max dapat dispesifikasi pada saat pembuatan Semafor

→ Fungsi yg dipakai adalah *CreateSemaphore*

→ Biasanya digunakan untuk membatasi jumlah *thread* yang memakai suatu resource secara bersamaan

# Semafor di dalam Pemrograman (2)

## ❖ Java™ Programming

- Semafor di Java™ bersifat **transparan** oleh programmer
- Java™ menyembunyikan Semafor dibalik konsep *monitor*
- *Reserved Word* yang dipakai Java™ adalah **synchronized**

# *Bounded-Buffer (1)*

- ❖ Pengertian:

  - Tempat penampung data yang ukurannya terbatas

- ❖ Contohnya:

  - Proses produsen dan konsumen

# *Bounded-Buffer (2)*

## ❖ Masalah produsen-konsumen

→ Produsen menaruh data pada buffer. Jika buffer tersebut sudah terisi penuh, maka produsen tidak melakukan apa-apa dan menunggu sampai konsumen mengosongkan isi buffer.

→ Konsumen mengambil data dari buffer. Jika buffer tersebut kosong, maka konsumen tidak melakukan apa-apa dan menunggu sampai buffer tersebut diisi oleh produsen.

# *Bounded-Buffer (3)*

- ❖ Solusi untuk memory yang dipakai bersama

```
#define BUFFER_SIZE 10
typedef struct {
...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

# *Bounded-Buffer (4)*

## ❖ Proses produsen

```
item nextProduced;
```

```
while (1) {  
    /* memproduksi sebuah item pada nextProduced  
    */  
    while (counter == BUFFER_SIZE) { ... do nothing  
    ... }  
    buffer[in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

# *Bounded-Buffer (5)*

❖ Proses konsumen

```
item nextConsumed;
```

```
while(1) {  
    while (counter==0) { ... do nothing ... }  
    nextConsumed = buffer(out);  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
}
```

# Masalah *Bounded-Buffer* (1)

- ❖ Jika produsen dan konsumen mengakses dan mengubah nilai counter secara bersamaan, maka nilai counter akan tidak sesuai
- ❖ Jadi, pernyataan **counter-** dan **counter++** harus dieksekusi secara atomic (tanpa terjadi interupsi)

# Masalah *Bounded-Buffer* (2)

Semafor yang dipakai adalah:

❖ Semafor *mutex*

→ Menyediakan *mutual exclusion* untuk mengakses *buffer*

→ Inisialisasi dengan nilai 1

❖ Semafor *full*

→ Menyatakan jumlah *buffer* yang sudah terisi

→ Inisialisasi dengan nilai 0

❖ Semafor *empty*

→ Menyatakan jumlah *buffer* yang kosong

→ Inisialisasi dengan nilai  $n$  (jumlah *buffer*)

# Masalah *Bounded-Buffer* (3)

❖ Inisialisasi

`full = 0, empty = n, mutex = 1`

# Masalah *Bounded-Buffer* (4)

❖ Proses produsen

```
do {
```

```
    ...
```

```
    memproduksi sebuah item pada nextp
```

```
    ...
```

```
    wait (empty);
```

```
    wait (mutex);
```

```
    ...
```

```
    menambahkan nextp ke buffer
```

```
    ...
```

```
    signal (mutex);
```

```
    signal (full);
```

```
} while (1);
```

# Masalah *Bounded-Buffer* (5)

❖ Proses konsumen

```
do {  
    wait (full)  
    wait (mutex);  
    ...  
    memindahkan sebuah item dari buffer ke next  
    ...  
    signal (mutex);  
    signal (empty);  
    ...  
    mengkonsumsi item di nextc  
    ...  
} while (1);
```

# Masalah *Readers/Writers*

## ❖ Definisi:

→ Diketahui dua macam proses:

- pembaca
- penulis

→ Kedua jenis proses berbagi sumber daya penyimpanan yang sama

Misal: Basis data

→ Tujuan: data tidak terkorupsi

→ Kondisi:

- Proses-proses pembaca dapat membaca sumber daya secara simultan
- Hanya boleh ada satu penulis menulis pada setiap saat
- Bila ada yang menulis, tidak boleh ada yang membaca

# Pembaca(1)

```
class Pembaca extends Thread
{
    public Pembaca(int id, Berkas berkas)
    {
        this.id = id;
        this.berkas = berkas;
        hitungan = 0;
        setName("Pembaca " + id);
    }

    public void run()
    {
        while (hitungan < PembuatBacaTulis1.BATAS)
        {
            berkas.tidur();
            System.out.println("Pembaca " + id + " tidur.");
            berkas.mulaiMembaca();
            System.out.println("Pembaca " + id + " mulai membaca.");
            berkas.tidur();
        }
    }
}
```

# Pembaca(2)

```
        System.out.println("Pembaca " + id + " sedang membaca.");
        berkas.selesaiMembaca();
        System.out.println("Pembaca " + id + " selesai membaca.");
        hitungan++;
    }
}

private Berkas berkas;
private int id;
private int hitungan;
}
```

# Penulis(1)

```
class Penulis extends Thread
{
    public Penulis(int id, Berkas berkas)
    {
        this.id = id;
        this.berkas = berkas;
        hitungan = 0;
        setName("Penulis " + id);
    }

    public void run()
    {
        while (hitungan < PembuatBacaTulis1.BATAS)
        {
            berkas.tidur();
            System.out.println("Penulis " + id + " tidur.");
            berkas.mulaiMenulis();
        }
    }
}
```

# Penulis(2)

```
        System.out.println("Penulis " + id + " mulai menulis.");
        berkas.tidur();
        System.out.println("Penulis " + id + " sedang menulis.");
        berkas.selesaiMenulis();
        System.out.println("Penulis " + id + " selesai menulis.");
        hitungan++;
    }
}

private Berkas berkas;
private int id;
private int hitungan;
}
```

# Solusi

❖ Tergantung prioritas:

→ Pembaca lebih diprioritaskan

→ Penulis lebih diprioritaskan

→ Pembaca dan penulis memiliki prioritas yang sama

# Solusi 1(1)

## ❖ Pembaca memiliki prioritas

- → Kendala: Bisa terjadi *starvation* untuk proses penulis

```
public void mulaiMembaca()
{
    mutex.tunggu();
    nPembaca++;
    if (nPembaca == 1) brks.tunggu();
    mutex.sinyal();
}
public void selesaiMembaca()
{
    mutex.tunggu();
    --nPembaca;
    if (nPembaca == 0) brks.sinyal();
    mutex.sinyal();
}
```

# Solusi 1(2)

```
public void mulaiMenulis()  
{  
    brks.tunggu();  
}
```

```
public void selesaiMenulis()  
{  
    brks.sinyal();  
}
```

# Solusi 2(1)

❖ Penulis memiliki prioritas

→Kendala: Bisa terjadi *starvation* untuk proses pembaca

```
public void mulaiMembaca()
{
    mutex1.tunggu();
    baca.tunggu();
    mutex2.tunggu();
    nPembaca++;
    if (nPembaca == 1) tulis.tunggu();
    mutex2.sinyal();
    baca.sinyal();
    mutex1.sinyal();
}
```

# Solusi 2(2)

```
public void selesaiMembaca()
{
    mutex2.tunggu();
    nPembaca--;
    if (nPembaca == 0) tulis.sinyal();
    mutex2.sinyal();
}
```

```
public void mulaiMenulis()
{
    mutex3.tunggu();
    nPenulis++;
    if (nPenulis == 1) baca.tunggu();
    mutex3.sinyal();
    tulis.tunggu();
}
```

# Solusi 2(3)

```
public void selesaiMenulis()
{
    tulis.sinyal();
    mutex3.tunggu();
    nPenulis--;
    if (nPenulis == 0) baca.sinyal();
    mutex3.sinyal();
}
```

# Solusi 3

❖ Penulis dan pembaca tidak ada memiliki prioritas

→Kendala: Bisa terjadi antrian yang panjang

# Masalah *Dining Philosophers*

## ❖ Definisi:

→ Diketahui:

- Pembaca
- Sebuah meja bundar
- N filsuf duduk melingkar di meja bundar
- Antara dua filsuf terdapat sebuah sumpit
- Didepan setiap filsuf terdapat semangkuk mie

## ❖ Setiap filsuf hanya dapat berada pada salah satu kondisi berikut:

- Berpikir
- Lapar
- Makan

# Masalah *Dining Philosophers*

❖ Pembuatan solusi

❖ Dua hal yang harus diperhatikan:

→ *Deadlock*: Semua filsuf ingin makan dan telah memegang sumpit

→ *Starvation*: Ada filsuf yang kelaparan dalam waktu yang lama

# Kelemahan Penggunaan Semafor

- ❖ Termasuk *Low Level*
- ❖ Kesulitan dalam pemeliharannya, karena tersebar dalam seluruh program.
- ❖ Menghapus *wait* → dapat terjadi *non-mutual exclusion*.
- ❖ Menghapus *signal* → dapat terjadi *deadlock*
- ❖ Salah meletakkan *code* → error!!!
- ❖ *Error* yang terjadi sulit untuk dideteksi
- ❖ Lebih baik menggunakan *high level construct*

# Konstruksi Sinkronisasi Tingkat Tinggi

❖ *Critical Region*

❖ Monitor

# *Critical Region*

- ❖ Definisi :

Bagian dari kode yang selalu dilaksanakan dalam kondisi mutual eksklusif.

- ❖ Biarkan *compiler* yang mengkondisikan mutual eksklusif!  
(bukan *programmer*).

# Komponen *Critical Region*

- ❖ *Shared variable*  $v$  dengan tipe  $T$  yang dengan deklarasi:

$v : \text{shared } T;$

yang hanya dapat diakses dalam kondisi mutual eksklusif.

- ❖ Pernyataan yang mengidentifikasi *critical region* di mana variabel diakses :

$\text{region } v \text{ when } (B) P;$

- ❖  $B$  adalah ekspresi boolean yang mengatur akses ke *critical region*.
- ❖  $P$  adalah pernyataan yang sedang dieksekusi, dimana proses lain tidak boleh mengakses  $v$ .

# Konsep Kerja *Critical Region* (1)

- ❖ Mutex adalah semacam kunci untuk mengakses wilayah yang mutual eksklusif
- ❖ Suatu proses yang ingin memasuki *critical region* harus mendapatkan mutex terlebih dahulu.
  - Tidak mendapat mutex → masuk dalam antrian utama
  - Mendapat mutex → tes nilai boolean B
- ❖ Nilai boolean B
  - **True**: proses berlanjut
  - **False**: melepaskan mutex dan masuk dalam antrian \kedua lalu ulang dari awal.

# Konsep Kerja *Critical Region* (2)

❖ Bentuk umum

```
region v when B do
begin
.....
end
```

❖ Menggunakan antrian jenis *FIFO*.

# Implementasi dalam *Bounded* *Buffer*

❖ Data yang diakses bersama-sama

```
struct buffer{  
    int pool[n];  
    int count, in, out;  
}
```

Proses yang dilakukan produsen

```
region buffer when (count < n){  
    pool[in] nextp;  
    in:= (in+1) % n;  
    count++  
}
```

# Implementasi dalam *Bounded Buffer* (2)

❖ Proses yang dilakukan konsumen

```
region buffer when (count>0){  
    nextc = pool[out];  
    out:= (out+1) % n;  
    count--;  
}
```

# Kelemahan *Critical Region*

- ❖ Lebih sulit untuk diimplementasi dibandingkan semafor
- ❖ Masih tersebar dalam kode program.
- ❖ Tidak ada kontrol terhadap manipulasi variabel yang diproteksi.

Bila sebuah proses memasuki *critical region*, ia dapat memanipulasi variabel yang diakses bersama-sama tersebut.

# Monitor (1)

- ❖ Diperkenalkan oleh Hoare (1974) dan Brinch Hansen
- ❖ Definisi:  
merupakan kumpulan dari prosedur, variabel, konstan dan struktur data dalam suatu modul. Proses dapat memanggil prosedur di dalam monitor, tetapi tidak dapat mengakses struktur data (variabel-variabel) internal dalam monitor dengan prosedur di luar monitor.
- ❖ Mengatasi manipulasi yang tidak sah atas variabel yang diakses bersama-sama karena variabel lokal hanya diakses prosedur lokal.

# Monitor (2)

- ❖ Hanya satu proses yang dapat aktif dalam monitor dalam suatu waktu.
- ❖ Antrian yang terbatas
- ❖ Agar lebih ampuh, harus menggunakan mekanisme sinkronisasi tambahan.  
Misalnya dengan menggunakan variabel kondisi yang terdapat dalam monitor dan hanya dapat diakses oleh monitor.

# Variabel Kondisi

## ❖ Deklarasi variabel kondisi

```
var  
c : condition;
```

Operasi yang dapat dilakukan terhadap variabel kondisi :

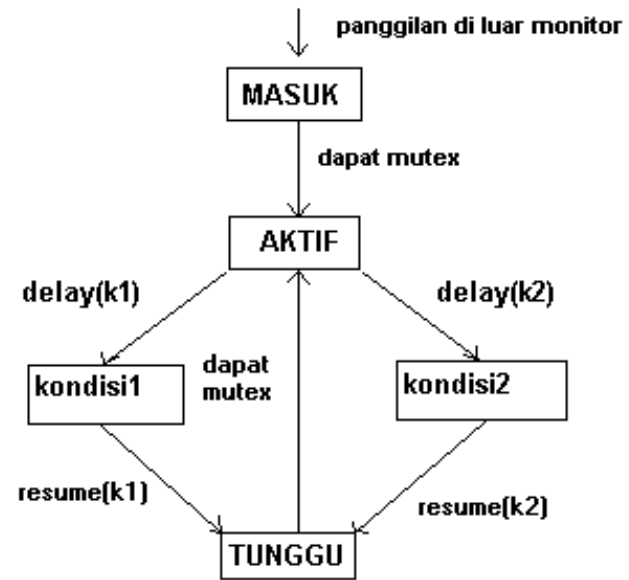
→ **delay** : menyerupai **wait** dalam semafor.

**delay(c)** membuat proses yang memanggil masuk dalam *block* dan melepaskan penguncian terhadap monitor.

→ **resume** : menyerupai **signal** dalam semafor.

**resume(c)** *unblock* proses yang sedang menunggu. resume tidak melakukan apa-apa bila delay tidak dipanggil (beda dengan *signal* dalam semafor).

# Struktur Monitor



# Implementasi Monitor (1)

## *Pseudocode*

```
monitor diningPhilosophers{
    int[] status = new int[5];
    boolean[] kiriLapar = new boolean[5];
    boolean[] kananLapar = new boolean[5];
    static final int BINGUNG = 0;
    static final int LAPAR = 1;
    static final int MAKAN = 2;
    condition[] aku = new condition[5];
    public diningPhilosophers{
        for (int i=0; i<5; i++){
            status[i] = BINGUNG;
            kiriLapar[i] = false;
            kananLapar[i] = false;
        }
    }
}
```

# Implementasi Monitor (2)

```
public entry ambil(int i){
    status[i] = LAPAR;
    cek(i);
    if (status[i] != MAKAN)
        aku[i].wait();
    kananLapar[kiri(i)] = false;
    kiriLapar[kanan(i)] = false;
}
public entry taruh(int i){
    status[i] = BINGUNG;
    cek(kiri(i));
    if (status[kiri(i)] == LAPAR)
        kiriLapar[i] = true;
    cek(kanan(i));
    if (status[kanan(i)] ==LAPAR)
        kananLapar[i] = true;
}
```

# Implementasi Monitor (3)

```
private cek(int i){
    if (status[kanan(i)] != MAKAN &&
        status[i] == LAPAR &&
        status[kiri(i)] != MAKAN &&
        !kiriLapar(i) && !kananLapar(i)){
        status[i] = MAKAN;
        aku[i].signal();
    }
}
private int kiri(int i){
    return (i+1)%5;
}
private int kanan(int i){
    return (i+4)%5;
}
```

# Kelemahan Monitor

- ❖ Tidak semua *compiler* dapat menerapkan peraturan mutual eksklusif seperti yang dibutuhkan oleh Monitor
- ❖ Tidak dapat diterapkan dalam sistem terdistribusi yang memiliki memori masing-masing

# *Deadlock (1)*

## ❖ Latar Belakang

- Jika proses 1 sedang menggunakan sumber daya 1 dan menunggu sumber daya 2 yang ia butuhkan, sedangkan proses 2 sedang menggunakan sumber daya 2 dan menunggu sumber daya 1
- Atau dengan kata lain saat proses masuk dalam status menunggu, ia tidak akan pernah keluar sebab sumber daya yang dibutuhkan sedang digunakan oleh proses lain yang sedang menunggu pula

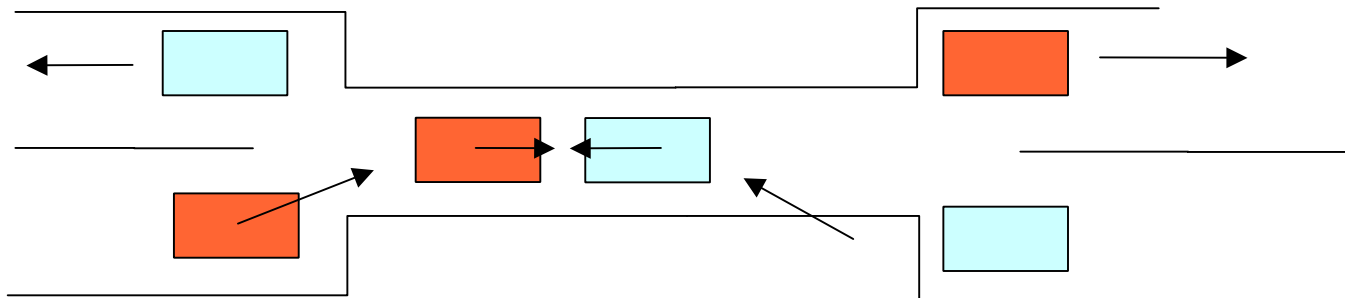
# *Deadlock (2)*

- ❖ Kemungkinan penyebab (menurut Coffman et al. (1971))
  - *Mutual Exclusion*: satu proses satu sumber daya
  - *Hold and Wait*: proses yang memegang sumber daya bisa meminta sumber daya lain
  - *No Preemption*: sumber daya yang sedang digunakan oleh suatu proses tidak bisa sembarangan diambil dari proses tersebut, melainkan harus dilepaskan dengan sendirinya oleh proses.
  - *Circular Wait* : setiap proses menunggu sumber daya dari proses berikutnya

# Contoh Persimpangan Jembatan

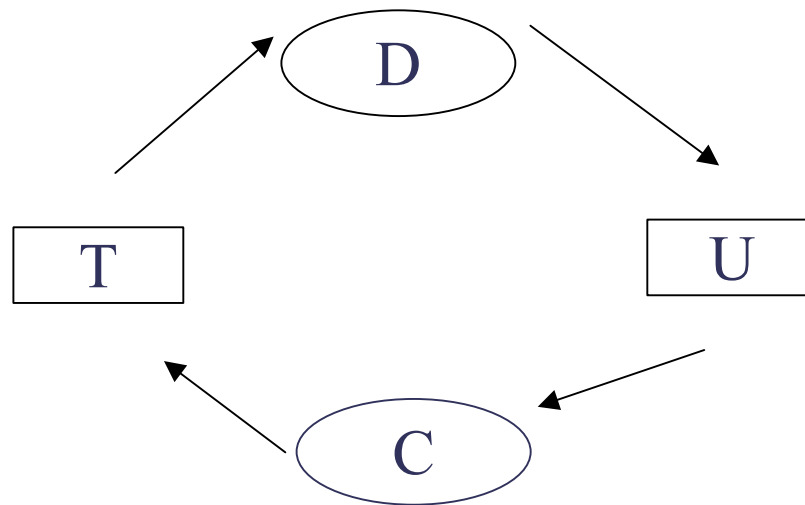
Pada satu jalan yang memungkinkan hanya satu arah yang berjalan

- Setiap jalan bisa dianggap sebagai sumber daya
- Saat *deadlock* terjadi hanya bisa diatasi jika salah satu mobil mundur, dalam hal ini butuh sumber daya yang direalokasikan
- Bahkan beberapa mobil harus mundur jika *deadlock* terjadi
- Pada kasus ini juga bisa terjadi “kelaparan”, yaitu ada proses yang tidak terlayani



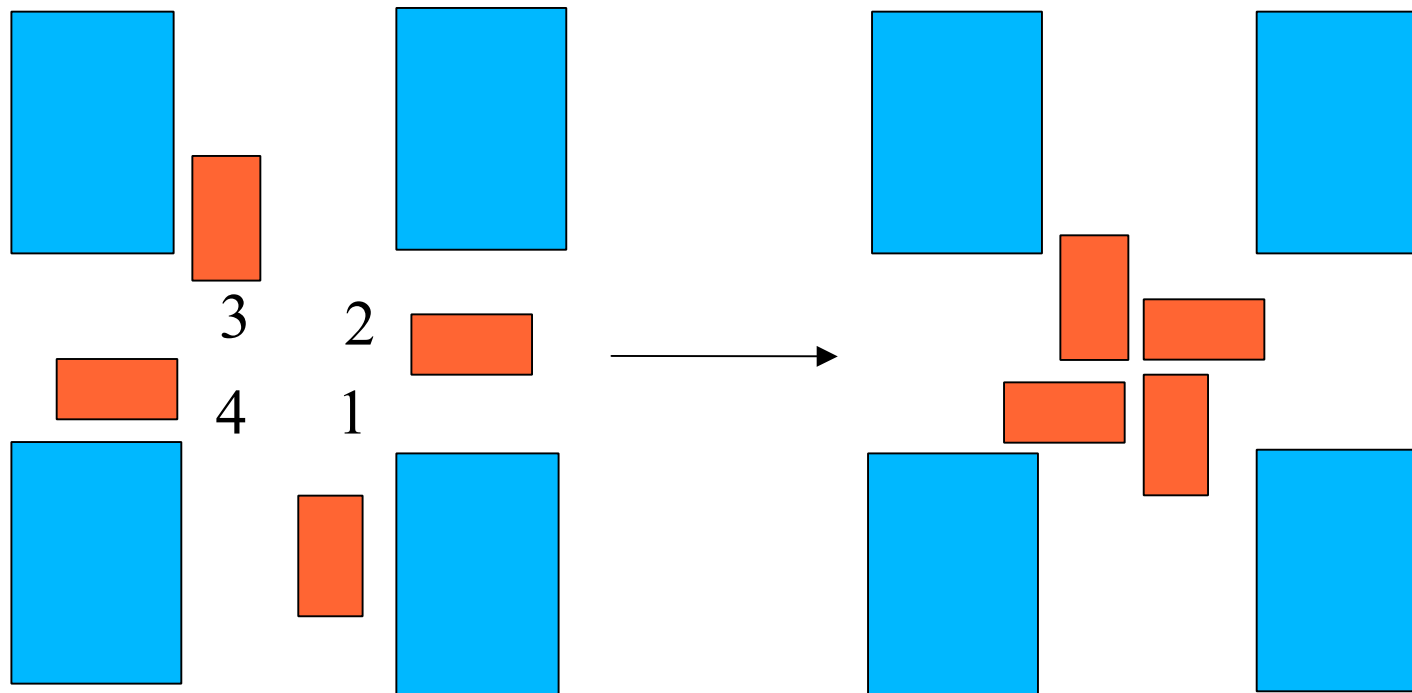
# Contoh *Deadlock* dalam Graf

Saat D membutuhkan sumber daya U, U sedang digunakan oleh C, demikian juga sebaliknya, sehingga terjadi *deadlock*



# Contoh Mobil di Persimpangan Jalan

- ❖ Dalam kasus ini, setiap mobil berjalan sesuai nomor yang ditentukan
- ❖ Tetapi pada akhirnya mereka akan bertemu pada suatu titik yang menyebabkan *deadlock*



# Mencegah *Deadlock* (1)

- ❖ *Mengatasi Mutual Exclusion* yaitu dengan cara tidak berbagi data dengan proses lain atau dengan kata lain menyediakan data sendiri
- ❖ Memegang dan menunggu yaitu menunggu sampai sumber daya yang akan digunakan tidak lagi digunakan oleh proses lain
  - Artinya proses dapat berlangsung jika semua sumber daya yang diperlukan tidak digunakan oleh proses lain
  - Dapat menyebabkan terjadinya “kelaparan” sebab ada proses yang tidak mendapat sumber daya sehingga menunggu terlalu lama

# Mencegah *Deadlock* (2)

- ❖ Mengatasi masalah *No Preemptive* dengan cara memerintahkan seluruh proses menunggu dan mempersilakan hanya proses yang memiliki sumber daya lama dan baru sesuai dengan daftar sumber daya yang sama dengan yang lain yang boleh berjalan
- ❖ Setiap kebutuhan total didata terlebih dahulu

# Menangani *Deadlock* (1)

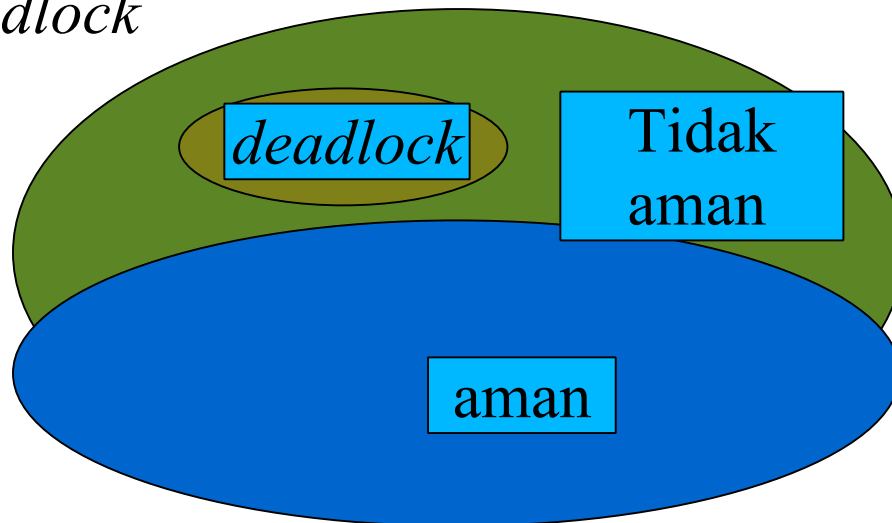
- ❖ Memakai protokol untuk menghindari atau mengabaikan *deadlock*, sehingga dipastikan tidak akan memasuki keadaan *deadlock*
  - *Deadlock Avoidance* => memerintahkan pada sistem operasi untuk memberi informasi tentang operasi mana yang bisa dan perlu dilaksanakan (keadaan aman). Selain itu bisa juga menggunakan algoritma bankir
  - *Deadlock Prevention* => memastikan bahwa keadaan yang penting tidak bisa menunggu
- ❖ Membiarkan sistem memasuki waktu *deadlock*, mendeteksinya, dan memperbaikinya
  - Algoritma mendeteksi *deadlock*
  - Algoritma memperbaiki *deadlock*

# Menangani *Deadlock* (2)

- ❖ Mengabaikan adanya *deadlock* dan menganggap keadaan *deadlock* tidak pernah terjadi ( Algoritma Ostrich )
  - Secara sederhana algoritma ini dapat dikatakan abaikan *deadlock* seakan-akan tidak ada masalah apapun dengannya
  - Algoritma ini disadur oleh Sistem Operasi Unix, meskipun memerlukan biaya yang cukup besar untuk mengatasi sebuah *deadlock*

# Kondisi Aman

- ❖ Saat sistem meminta izin untuk mengambil sumber dayanya, sistem operasi harus memastikan bahwa ia dalam kondisi aman
- ❖ Sistem dalam kondisi aman jika seluruh sistem dapat berjalan tanpa terancam kekurangan sumber daya atau *deadlock*



# Algoritma Bankir

- ❖ Setiap proses yang masuk harus memberitahu berapa banyak sumber daya maksimum yang dibutuhkan
- ❖ Setelah itu sistem mendeteksi apakah sumber daya yang dibutuhkan memang bisa dijalankan dalam kondisi aman
  - Jika ya, maka sistem akan melepaskan sumber dayanya untuk digunakan
  - Jika tidak, maka proses harus menunggu hingga sumber dayanya cukup

# Keluar dari *Deadlock*

## ❖ Mematikan program

- Matikan semua proses yang berjalan
- Hanya matikan proses yang berjalan dalam siklus *deadlock*

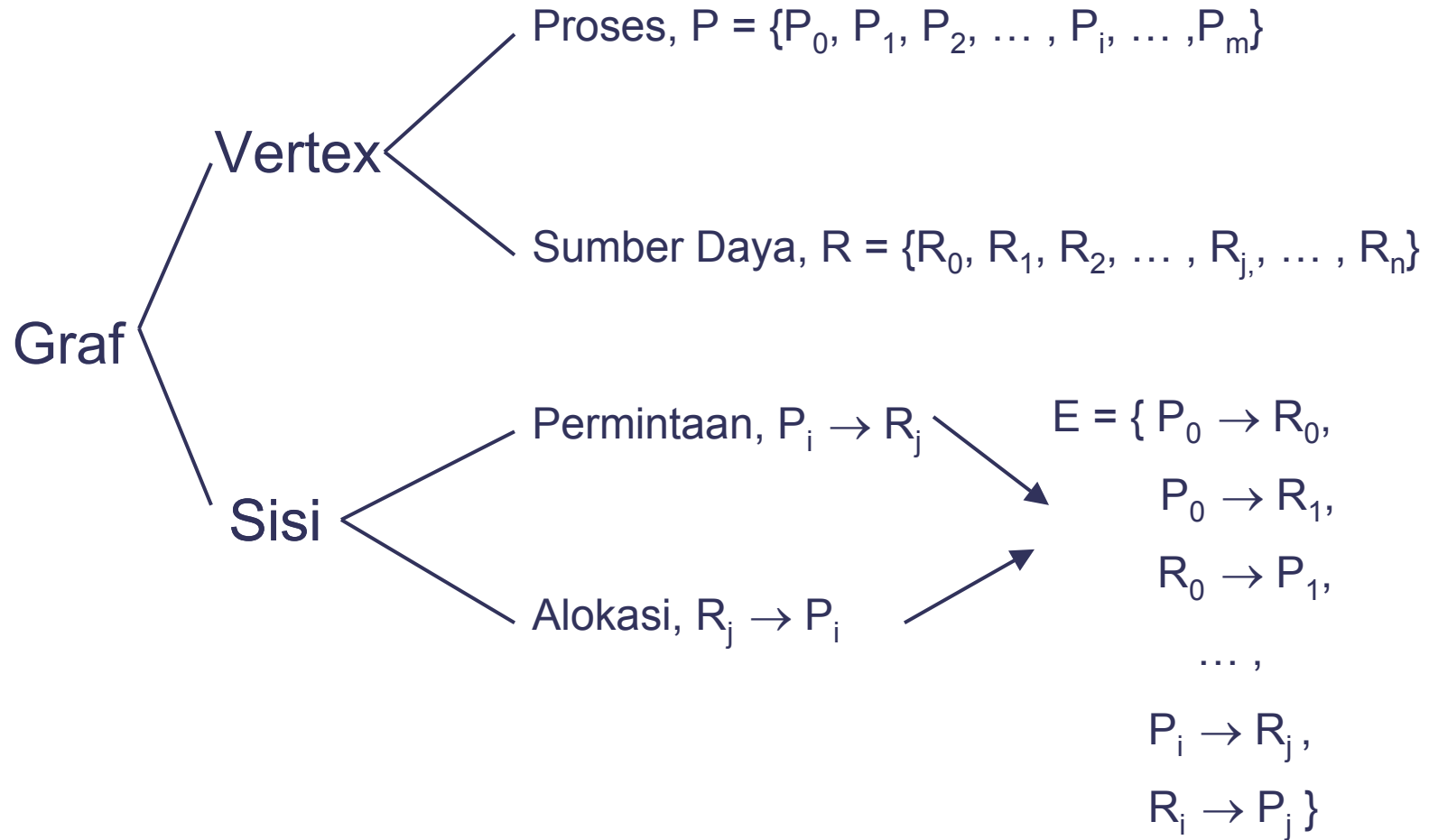
Urutan terminasinya:

- Prioritas rendah
- Seberapa jauh dan berapa banyak bahan yang proses sudah atau akan dibutuhkan atau dilakukan

## ❖ Data sumber daya yang dibutuhkan sebelumnya

- Pilih korban proses dengan sumber daya terkecil
- Kembali ke keadaan aman, jika *deadlock* sudah terdeteksi
- Resiko yang harus dihadapi ialah proses dengan sumber daya terkecil akan mengalami kelaparan atau tidak pernah dieksekusi

# Diagram Graf



# Komponen Graf Alokasi Sumber Daya (1)

❖ Himpunan Vertex, dibagi menjadi 2 bagian:

→ Proses,  $P = \{P_0, P_1, P_2, \dots, P_j, \dots, P_n\}$

Terdiri dari semua proses yang ada di sistem.

Digambarkan sebagai:



# Komponen Graf Alokasi Sumber Daya (2)

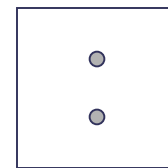
❖ Sumber Daya,  $R = \{R_0, R_1, R_2, \dots, R_m\}$

Terdiri dari semua sumber daya yang tersedia di sistem.

Dalam hal ini jumlah proses dan sumber daya TIDAK SELALU sama.

Digambarkan beserta jumlah instans yang tersedia.

Contoh  $R_j$  dengan 2 instans:



$R_j$

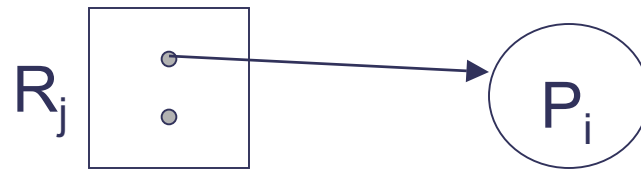
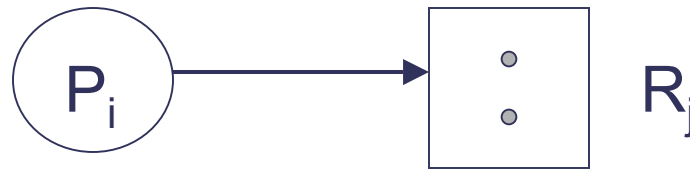
# Komponen Graf Alokasi Sumber Daya (3)

❖ Himpunan Sisi, dibagi menjadi 2 bagian:

→ Sisi Permintaan,  $P_i \rightarrow R_j$

Menggambarkan proses  $P_i$  yang meminta sumber daya

$R_j$ .

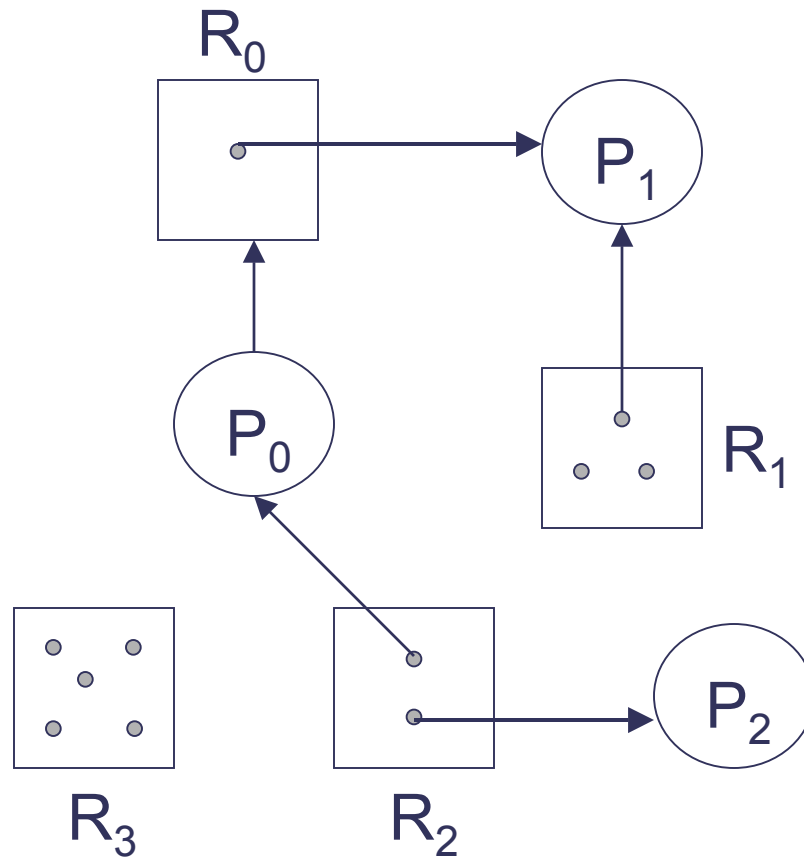


→ Sisi Alokasi,  $R_j \rightarrow P_i$

Menggambarkan sumber daya  $R_j$  yang mengalokasikan

SALAH SATU instansnya pada proses  $P_i$ .

# Contoh Graf Alokasi Sumber Daya (1)



# Contoh Graf Alokasi Sumber Daya (2)

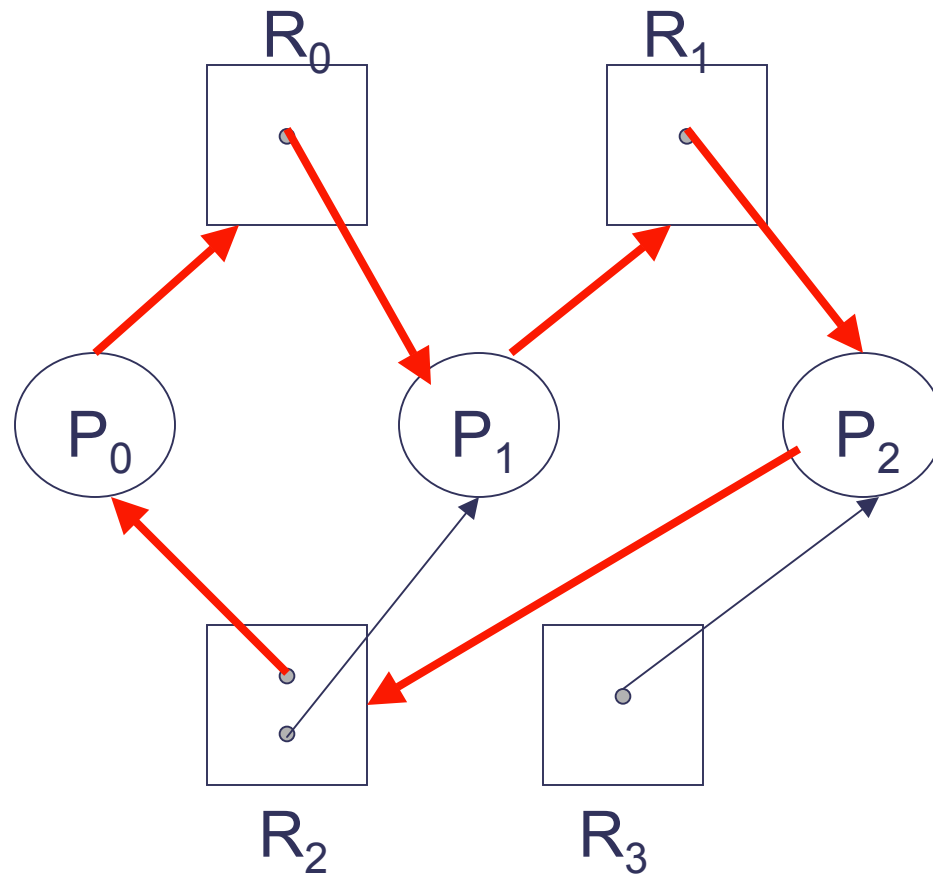
$$V = \{ P_0, P_1, P_2, R_0, R_1, R_2, R_3 \}$$

$$E = \{ P_0 \rightarrow R_0, R_0 \rightarrow P_1, R_1 \rightarrow P_1, R_2 \rightarrow P_0, R_2 \rightarrow P_2 \}$$

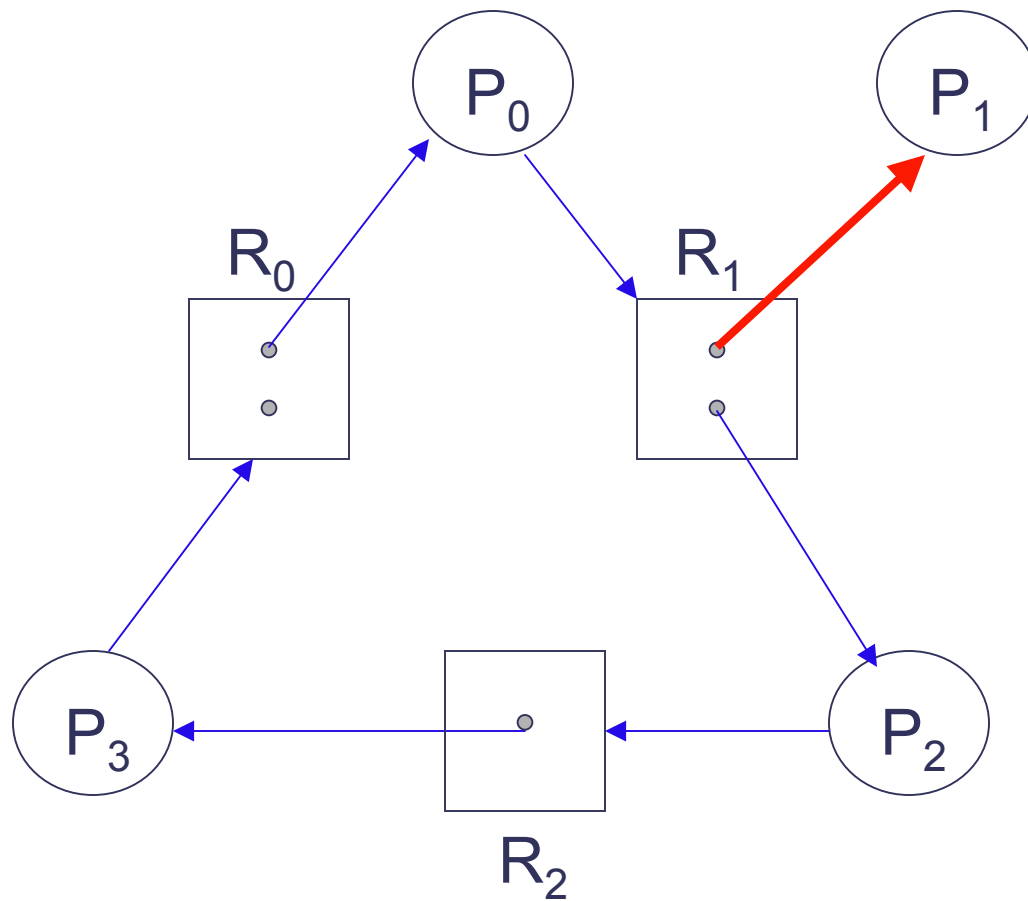
Keterangan:

- $P_0$  meminta sumber daya dari  $R_0$
- $R_0$  memberikan sumber dayanya kepada  $P_1$
- $R_1$  memberikan salah satu instans sumber dayanya kepada  $P_1$
- $R_2$  memberikan salah satu instans sumber dayanya kepada  $P_0$
- $R_2$  memberikan salah satu instans sumber dayanya kepada  $P_2$

# Graf Alokasi Sumber Daya dengan *Deadlock*



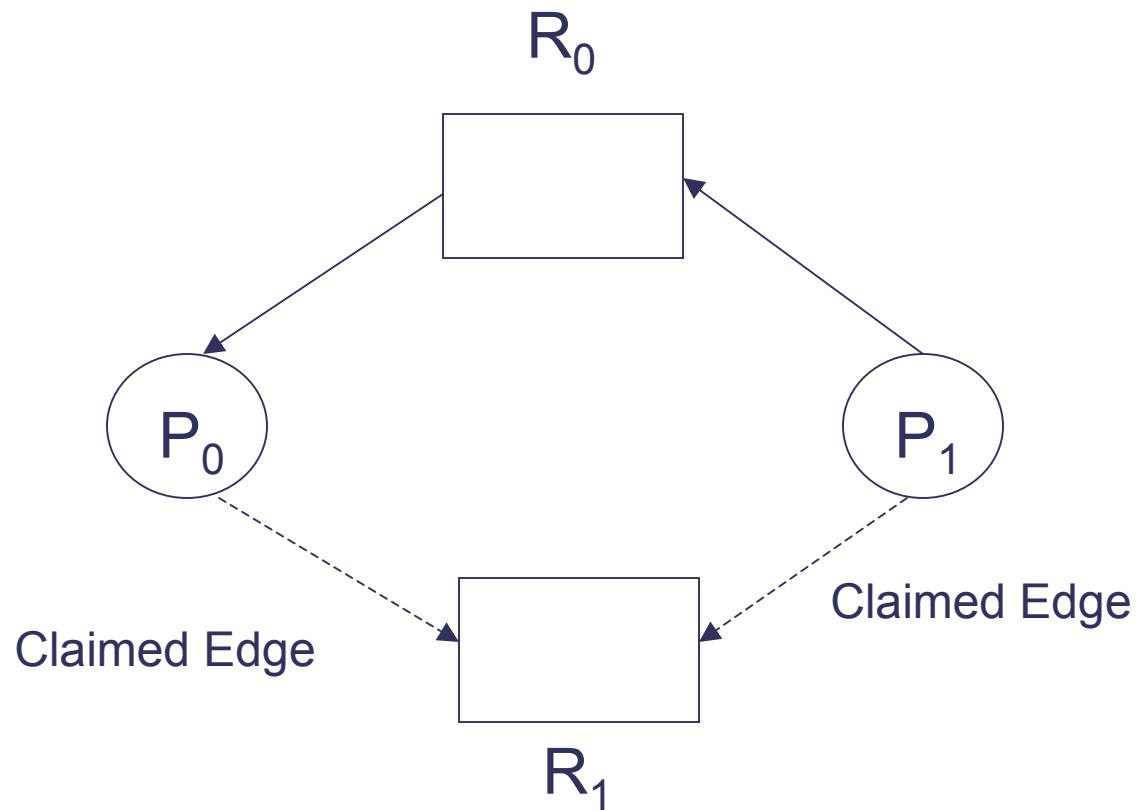
# Graf Alokasi Sumber Daya tanpa *Deadlock*



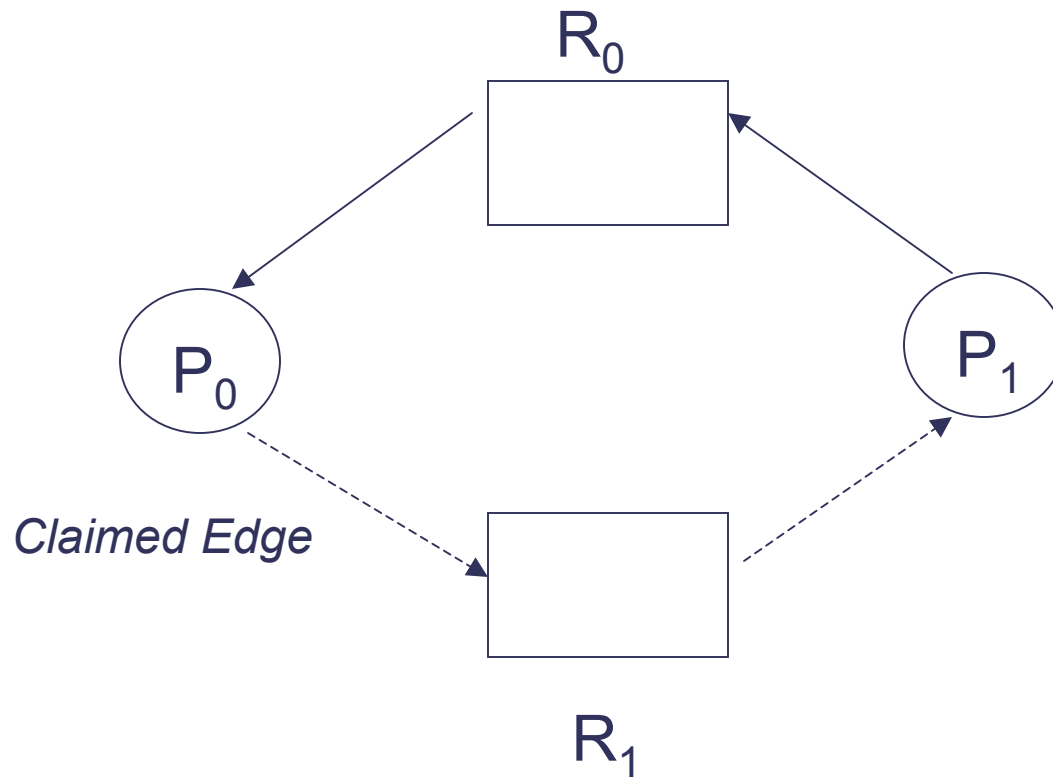
# Algoritma Graf Alokasi Sumber Daya

- ❖ *Claimed Edge*  $P_i \rightarrow R_j$  yang menggambarkan ada proses  $P_j$  yang juga meminta sumber daya  $R_j$ ; kemudian sisi tersebut diubah menjadi garis putus-putus.
- ❖ *Claimed Edge* diubah menjadi sisi permintaan ketika ada proses yang memerlukan sumber daya.
- ❖ Ketika suatu sumber daya dilepaskan oleh suatu proses, sisi pengalokasian diubah *Claimed Edge*.
- ❖ Sumber daya harus dinyatakan sebagai “*a priori*” dalam suatu sistem.

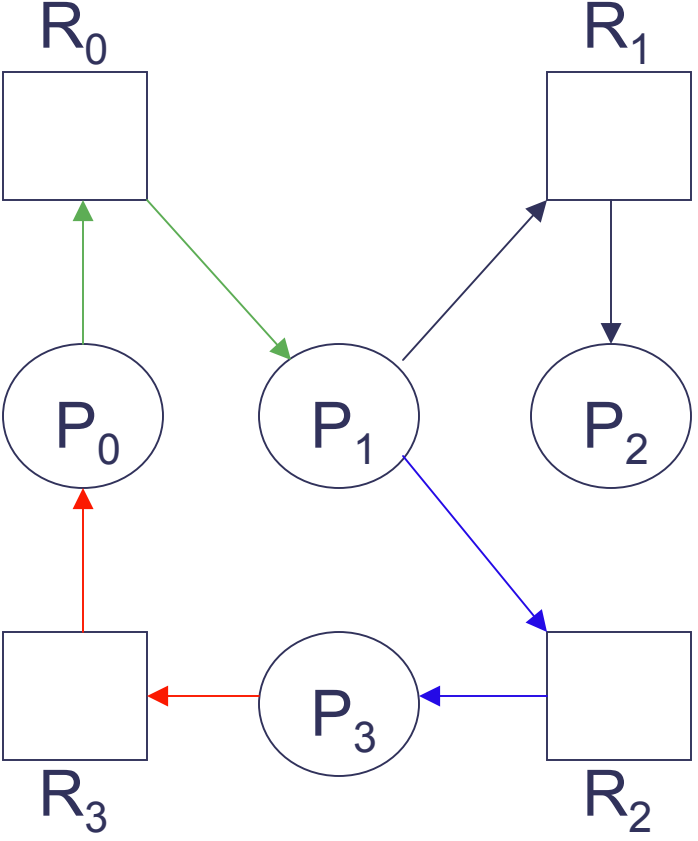
# Graf Alokasi Sumber Daya untuk Pencegahan *Deadlock*



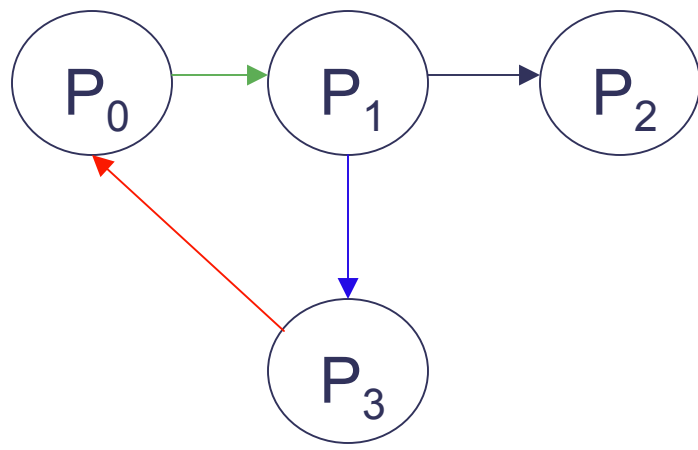
# Kondisi Tidak Aman Graf Alokasi Sumber Daya



# Graf Alokasi Sumber Daya dan Graf Tunggu



Graf Alokasi Sumber Daya



Graf Tunggu