SISTEM OPERASI

PERTEMUAN VI : SINKRONISASI

KOMUNIKASI PROSES

- Sistem Berbagi Memori ➔ Mengalokasikan suatu alamat memori untuk dipakai berkomunikasi antar proses.
- Sistem Berkirim Pesan ➔ Membagi variabel yang dibutuhkan. Proses ini menyediakan dua operasi yaitu mengirim pesan dan menerima pesan.
  - Komunikasi langsung
  - Komunikasi tidak langsung

KOMUNIKASI PROSES

- Komunikasi langsung → proses yang ingin berkirim pesan harus mengetahui secara jelas dengan siapa mereka berkirim pesan. Hal ini dapat mencegah pesan salah terkirim ke proses yang lain.
- Komunikasi tidak langsung → menggunakan sejenis kotak surat atau port yang mempunyai ID unik untuk menerima pesan. Proses dapat berhubungan satu sama lain jika mereka membagi port mereka

KOMUNIKASI PROSES

Komunikasi Langsung	Komunikasi Tidak Langsung
Link dapat otomatis dibuat	Link terbentuk jika beberapa proses membagi kotak surat mereka
Sebuah link terhubung tepat satu proses komunikasi berpasangan	Sebuah link dapat terhubung dengan banyak proses
Diantara pasangan itu terdapat tepat satu link	Setiap pasang proses dapat membagi beberapa link komunikasi
Link tersebut biasanya merupakan link komunikasi dua arah	Link merupakan link terarah ataupun link yang tidak terarah

CONCURRENCY (kebersamaan)

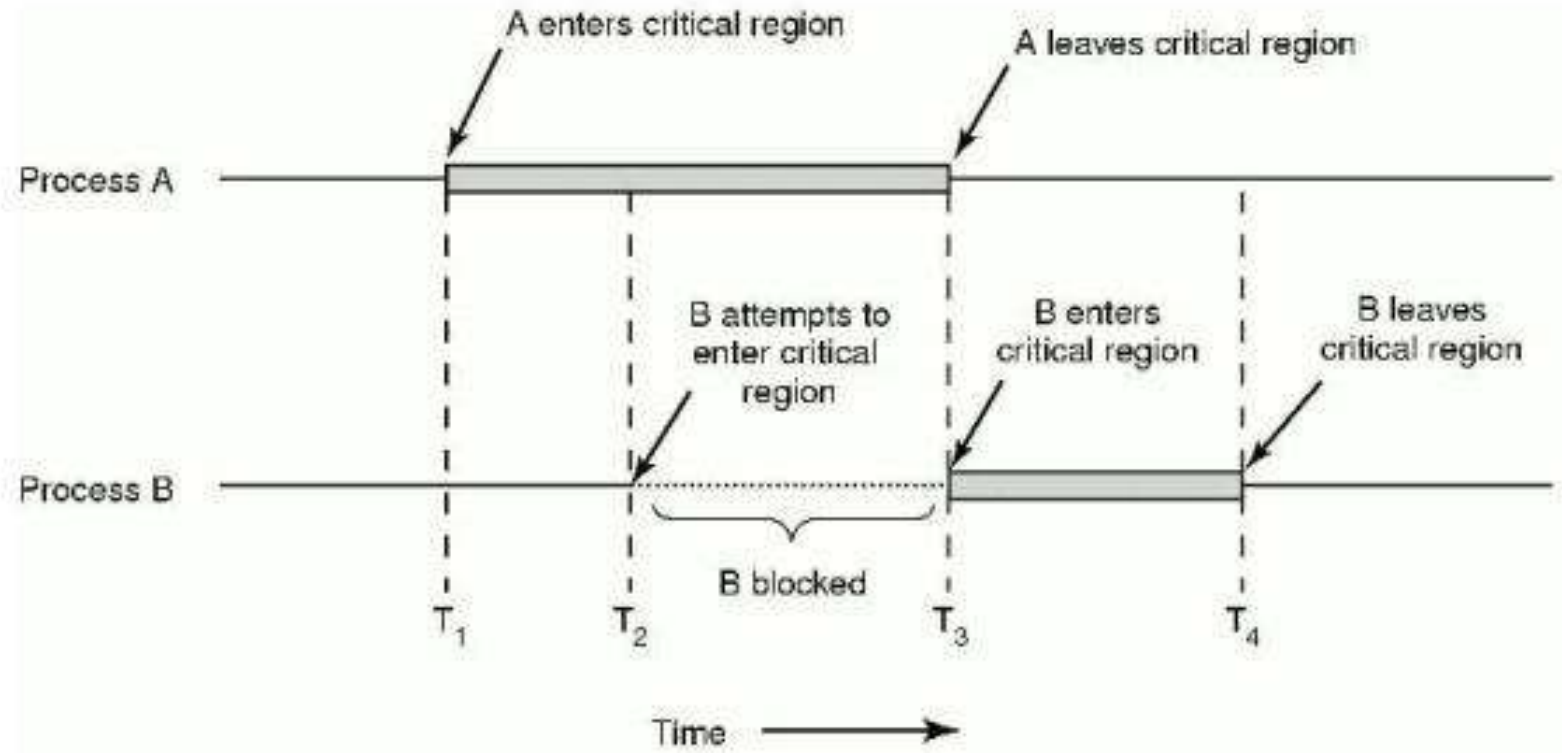
- Konkurensi merupakan landasan umum perancangan sistem operasi
- Proses, Penjadwalan, dan Sinkronisasi, merupakan 3 hal yang tidak bisa dipisahkan
- Konkurensi menjadi penting karena saat ini hampir seluruh sistem sudah mendukung *multiprogramming* ataupun *multithreading* yang mengharuskan adanya proses – proses yang bersifat konkuren.

CONCURRENCY (kebersamaan)

Terdapat beberapa **Tantangan** yang **harus diselesaikan** :

- ***Mutual Exclusion*** ➔ Merupakan kondisi dimana hanya ada satu proses yang mengakses suatu sumber daya. Bagian dari *code* program yang sedang mengakses sumber daya yang di pakai bersama dinamakan ***critical section***.
- *Deadlock*
- *Starvation*
- *Sinkronisasi*

Mutual Exclution

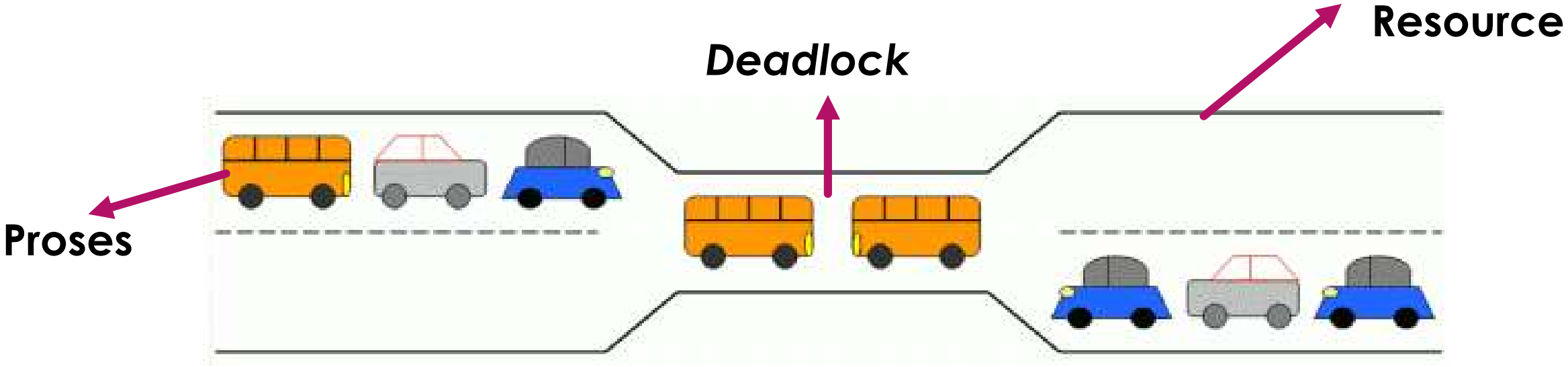


CONCURRENCY (kebersamaan)

Terdapat beberapa **Tantangan** yang **harus diselesaikan** :

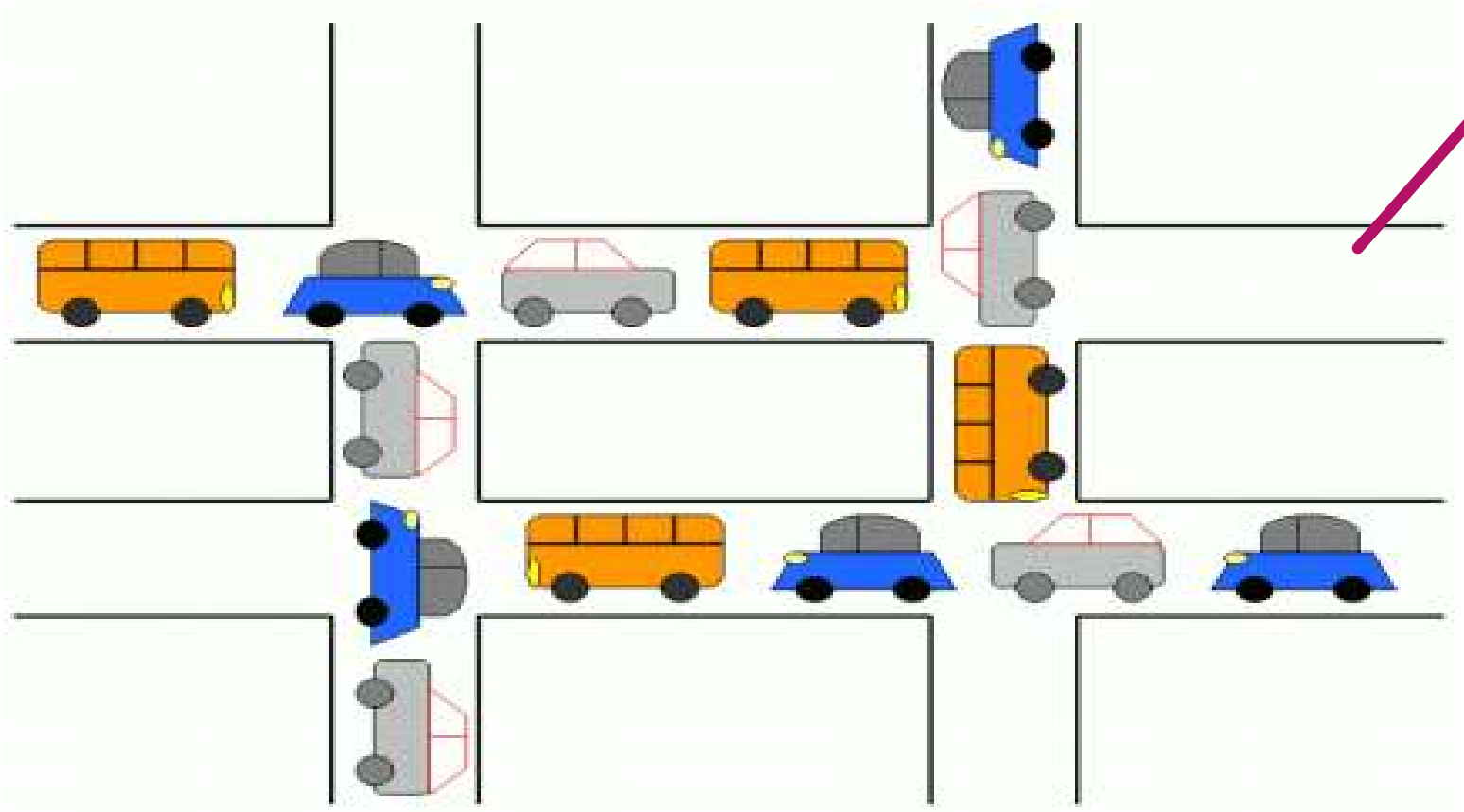
- *Mutual Exclusion*
- ***Deadlock*** ➔ keadaan dimana sistem seperti terhenti dikarenakan setiap proses memiliki sumber daya yang tidak bisa dibagi dan menunggu untuk mendapatkan sumber daya yang sedang dimiliki oleh proses lain.
- *Starvation*
- *Sinkronisasi*

Deadlock



Deadlock

Resource



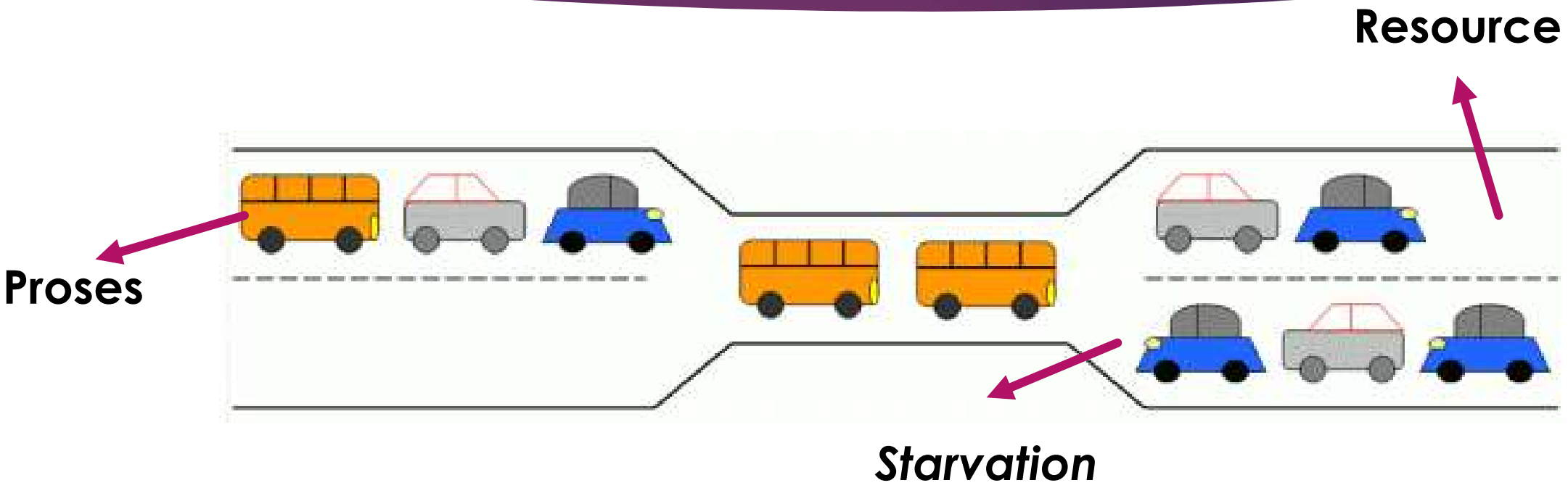
Proses

CONCURRENCY (kebersamaan)

Terdapat beberapa **Tantangan** yang **harus diselesaikan** :

- *Mutual Exclusion*
- *Deadlock*
- ***Starvation*** ➔ Proses yang kekurangan *resource* (karena terjadi *deadlock*) tidak akan pernah mendapat *resource* yang dibutuhkan sehingga mengalami *starving* (kelaparan).
- *Sinkronisasi*

STARVATION



CONCURRENCY (kebersamaan)

Terdapat beberapa **Tantangan** yang **harus diselesaikan** :

- *Mutual Exclusion*
- *Deadlock*
- *Starvation*
- ***Sinkronisasi*** ➔ menjaga agar data tersebut tetap konsisten dan mengatur urutan jalannya proses-proses sehingga dapat berjalan dengan lancar dan terhindar dari deadlock atau starvation.

PRINSIP – PRINSIP DAN LINGKUP *CONCURRENCY*

- Alokasi layanan pemroses untuk setiap proses
- Pemakaian bersama dan persaingan untuk mendapatkan sumber daya
- Komunikasi antar proses
- Sinkronisasi aktivitas dari banyak proses

PERMASALAHAN DALAM *CONCURRENCY*

- Kecepatan eksekusi proses – proses di sistem tidak dapat diprediksi.
  - Aktivitas – aktivitas proses lain
  - Cara sistem operasi menangani interrupt
  - Penjadwalan yang dilakukan oleh sistem operasic

TUGAS SISTEM OPERASI ➔ *CONCURRENCY*

- Mengetahui proses – proses yang sedang aktif
- Alokasi dan dealokasi beragam *resource* untuk tiap proses
- Proteksi data atau resource dari interferensi proses lain

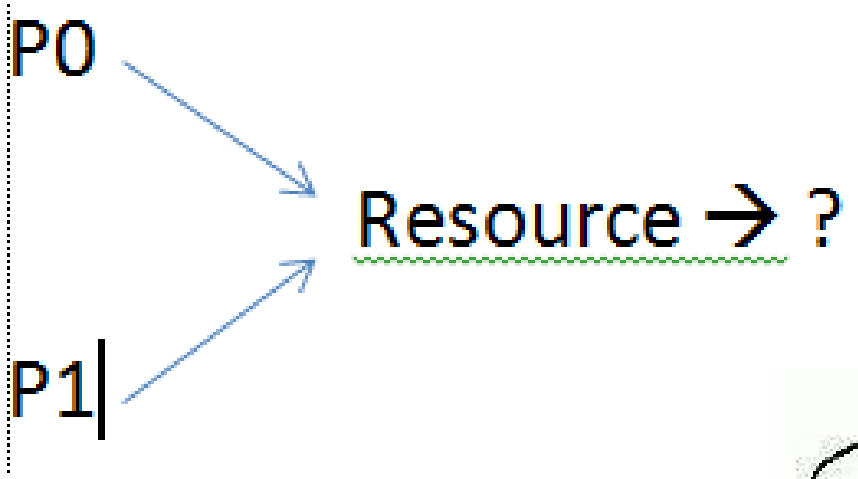
INTERAKSI ANTAR PROSES

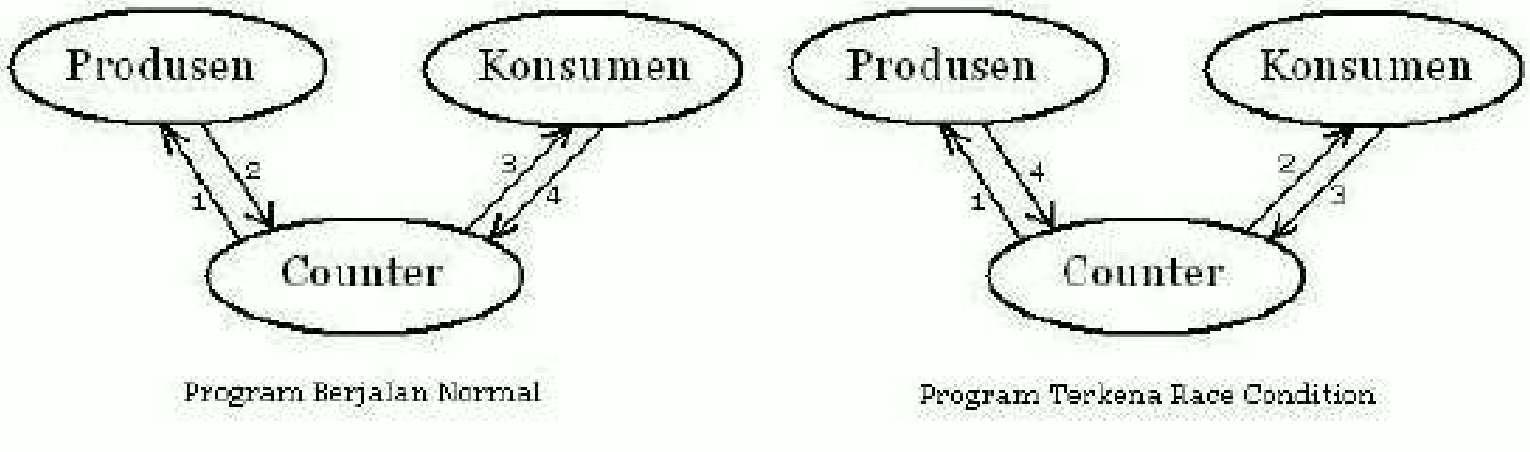
- *Race condition*
- *Critical section*
- *Mutual Exclusion with busy waiting*
- *Semaphore*
- *Monitor*
- *Message Passing*

RACE CONDITION

- *Race Condition* adalah suatu kondisi dimana dua atau lebih proses mengakses sumber daya / *resource* secara konkuren, hasil akhir dari data tersebut tergantung dari proses mana yang terakhir selesai dieksekusi.
- Untuk mencegah *race condition*, proses – proses yang berjalan secara konkuren harus disinkronisasi.

RACE CONDITION





Program Berjalan Normal

Program Terkena Race Condition

CRITICAL SECTION

➔ Solusi *Critical Section*, harus memenuhi **3 syarat** berikut:

- *Mutual Exclution* → Kondisi dimana Tidak ada dua proses yang menjalankan *critical section* bersamaan.
- Terjadi Kemajuan (*Progress*) → Proses yang sedang menjalankan *Remainder Section*-nya, tidak boleh menjalankan *Critical Section* berikutnya sebelum proses lain menyelesaikan *Critical Section*-nya.
- Ada Batas Waktu (*Bounded Waiting*) → Ada batas waktu suatu proses dapat menjalankan *Critical Section*-nya.

CRITICAL SECTION

➔ Solusi *Critical Section* :

- Perangkat Lunak → Menggunakan algoritma – algoritma untuk mengatasi masalah *Critical Section*.
- Perangkat Keras → Bergantung pada beberapa instruksi mesin tertentu, misalnya dengan me-non-aktifkan interupsi, mengunci suatu variabel tertentu atau menggunakan instruksi level mesin seperti tes dan set *(TestAndSet())*.

CRITICAL SECTION ➔ Perangkat Lunak

➔ Sinkronisasi 2 Proses :

- Algoritma Turn.
- Algoritma Flag ➔ Algoritma Dekker.
- Algoritma Turn-Flag ➔ Algoritma Peterson

➔ Sinkronisasi Banyak Proses :

- Algoritma Bakery (Algoritma Tukang Roti)

ALGORITMA TURN

- Algoritma *Turn* menerapkan sistem bergilir kepada kedua proses yang ingin mengeksekusi *Critical Section*, sehingga kedua proses tersebut harus bergantian menggunakan *Critical Section*.
- Algoritma ini menggunakan variabel bernama ***TURN***, nilai *turn* menentukan proses mana yang boleh memasuki *Critical Section*.
- Hanya proses yang mempunyai ID yang sama dengan ID giliran (*Turn*) yang boleh masuk ke *Critical Section*.

ALGORITMA TURN ➔ Ilustrasi

1. Diasumsikan terdapat 2 proses : P0 dan P1
2. Pada awalnya **variable *turn* diinisialisasi 0**,

   artinya **P0** yang **boleh** mengakses ***Critical Section*.**

   Jika ***turn*= 0** dan **P0 ingin menggunakan *Critical Section***, maka ia dapat mengakses *Critical Section*-nya.

3. Setelah selesai mengeksekusi *Critical Section*, **P0 akan mengubah *turn* menjadi 1**

   artinya giliran **P1 tiba** dan **P1 diperbolehkan** mengakses ***Critical Section*.**

   Ketika *turn*= 1 dan P0 ingin menggunakan *Critical Section*,

   maka **P0 harus menunggu** sampai **P1 selesai** menggunakan *Critical Section* dan **mengubah variable *turn*** menjadi **0**.

ALGORITMA TURN ➔ Masalah

- Pada algoritma ini masalah muncul ketika ada proses yang mendapat giliran memasuki *Critical Section* tapi tidak menggunakan gilirannya sementara proses yang lain ingin mengakses *Critical Section*.
- Misalkan ketika *turn*= 1 dan P1 tidak menggunakan gilirannya maka *turn* tidak berubah dan tetap 1.
- Kemudian P0 ingin menggunakan *Critical Section,*
- Maka ia harus menunggu sampai P1 menggunakan *Critical Section* dan mengubah *turn* menjadi 0.

ALGORITMA FLAG ➔ ALGORITMA DEKKER

- Algoritma Flag mengantisipasi masalah yang muncul pada Algoritma Turn dengan mengubah penggunaan variabel *turn* dengan variabel *flag*.
- Variabel *flag* menyimpan kondisi proses mana yang boleh masuk *Critical Section*.
- Setiap proses mengecek status proses yg lain. Jika proses lain sedang berada di *Critical Section*-nya, maka dia akan menunggu sampai proses lain tersebut keluar dari *Critical Section*-nya.
- Proses yang membutuhkan akses ke *Critical Section* akan memberikan nilai *flag*-nya ***true***. Sedangkan proses yang tidak membutuhkan *Critical Section* akan men-*setting* nilai *flag*-nya bernilai ***false***.

ALGORITMA FLAG ➔ Ilustrasi

1. Awalnya ***flag*** untuk kedua proses diinisialisai bernilai ***false***,
   artinya kedua proses tersebut **tidak membutuhkan *Critical Section*.**
2. Jika **P0 ingin mengakses** *Critical Section*, **P0 akan mengubah *flag*[0] menjadi *true***.
   Kemudian **P0** akan **mengecek apakah P1** juga **membutuhkan *Critical Section***, **jika *flag*[1] bernilai *false*** maka **P0 bisa menggunakan *Critical Section***.
3. Namun jika ***flag*[1] bernilai *true*** maka **P0 harus menunggu P1 menggunakan *Critical Section*** dan **mengubah *flag*[1]** menjadi ***false*.**

ALGORITMA FLAG ➔ Masalah

- Kedua proses tersebut akan men-set masing-masing *flag*-nya menjadi true (Ketika bersamaan).
  - P0 men- set *flag*[0] = *true* ; P1 men- set *flag*[1] = *true*.
- Kondisi ini menyebabkan kedua proses yang membutuhkan *Critical Section* tersebut akan saling menunggu dan "saling mempersilahkan" proses lain untuk mengakses *Critical Section*, akibatnya malah tidak ada yang mengakses *Critical Section*.

ALGORITMA TURN-FLAG ➔ ALGORITMA PETERSON

- Merupakan penggabungan antara Algoritma *Turn* dan Algoritma *Flag*.
- Sama seperti pada Algoritma *Turn* dan *Flag*, variabel *turn* menunjukkan giliran proses mana yang diperbolehkan memasuki *Critical Section* dan variable *flag* menunjukkan informasi proses kebutuhan akses ke *Critical Section*.

ALGORITMA TURN-FLAG ➔ ALGORITMA PETERSON → Ilustrasi

1. Awalnya *flag* untuk kedua proses diinisialisai bernilai ***false***,

   artinya **kedua proses** tersebut **tidak membutuhkan** akses ke *Critical Section*.
2. Kemudian jika suatu proses **ingin memasuki *Critical Section***, Proses tersebut akan **mengubah *flag*-nya menjadi true** lalu proses tersebut memberikan *turn* kepada lawannya.
3. Jika proses lain ***flag*-nya bernilai *false*,**

   maka **proses tersebut dapat menggunakan *Critical Section***, dan **setelah selesai menggunakan *Critical Section*** proses tersebut akan mengubah ***flag*-nya** menjadi ***false*.**
4. Tetapi apabila **proses lain** memiliki ***flag*-nya bernilai *true*** maka **proses tersebut-lah yang dapat menggunakan *Critical Section*,** dan **proses pertama harus menunggu** sampai proses lain menyelesaikan *Critical Section* dan **mengubah *flag*-nya menjadi *false***.

ALGORITMA TURN-FLAG ➔ Masalah

Bagaimana bila kedua proses membutuhkan *Critical Section* secara bersamaan?

Proses mana yang dapat mengakses *Critical Section* terlebih dahulu?

ALGORITMA TURN-FLAG ➔ Masalah

- Apabila kedua proses (P0 dan P1), menset masing-masing flag menjadi **true**

flag[0] = true
flag[1] = true

P0 dapat mengubah turn = 1
P1 juga dapat mengubah turn = 0

Proses mana yang* dapat mengakses *critical section terlebih dahulu?

proses yang terlebih dahulu mengubah *turn* menjadi *turn lawannya*

Algoritma ini memenuhi 3 syarat dari critical section : **Mutual exclution, Progress, dan bounded waiting**

ALGORITMA BAKERY ➔ ALGORITMA TUKANG ROTI

- Algoritma Tukang Roti adalah solusi untuk masalah *Critical Section* pada sejumlah n proses. Algoritma ini juga dikenal sebagai *Lamport's Baker Algorithm*.
- Ide algoritma ini adalah dengan menggunakan prinsip penjadwalan seperti yang ada di tempat penjualan roti.
- Para pelanggan yang ingin membeli roti sebelumnya harus mengambil nomor urut terlebih dahulu dan urutan orang yang boleh membeli ditentukan oleh nomor urut yang dimiliki masing-masing pelanggan tersebut.

ALGORITMA BAKERY ➔ ALGORITMA TUKANG ROTI

- Prinsip algoritma ini untuk menentukan proses yang boleh mengakses *Critical Section* sama seperti ilustrasi tukang roti diatas.
- Proses diibaratkan pelanggan yang jumlahnya n dan tiap proses yang membutuhkan *Critical Section* diberi nomor yang menentukan proses mana yang diperbolehkan untuk masuk kedalam *Critical Section*.

PERANGKAT SINKRONISASI

- Semaphore :

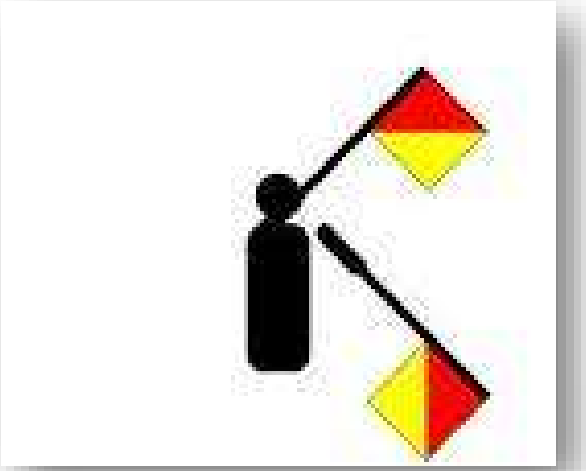
  "Sistem sinyal yang digunakan untuk berkomunikasi secara visual."

  "Sebuah variabel bertipe *integer* yang selain saat inisialisasi, hanya dapat diakses melalui dua operasi standar, yaitu *increment* dan *decrement*."

- Monitor :

  "suatu tipe data abstrak yang dapat mengatur aktivitas serta penggunaan resource oleh beberapa *thread*."

SEMAPHORE



SEMAPHORE

- Operasi standar dalam semaphore :
  - Proberen : *Test, Decrement, Release*, Buka, dll.
  - Verhogen : *Increment, Acquire*, Kunci, dll.

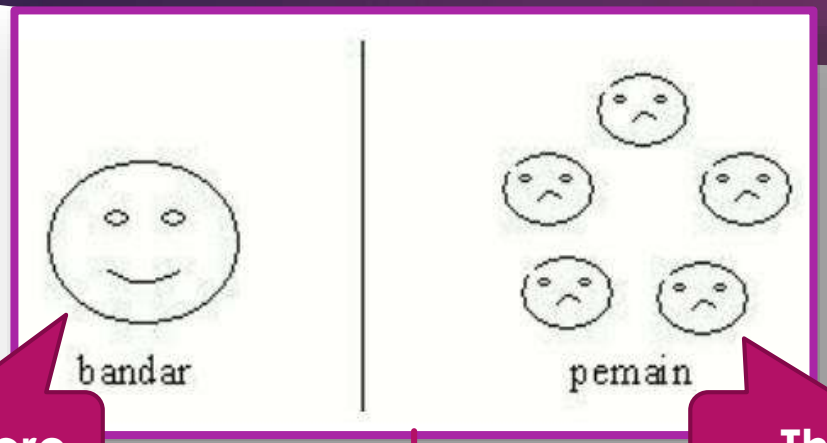
```
void kunci(int sem_value) {
    while(sem_value <= 0);
    sem_value--;
}
```

```
void buka(int sem_value) {
    sem_value++;
}
```

SEMAPHORE

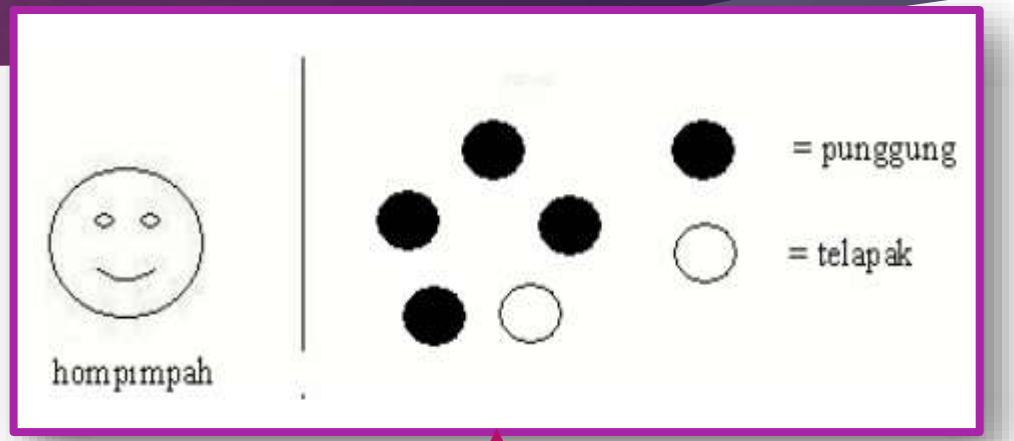
- Fungsi semaphore :
  - Menangani *mutual exclution*.
  - Sebagai *Resource Controller*.
  - Komunikasi antar proses.

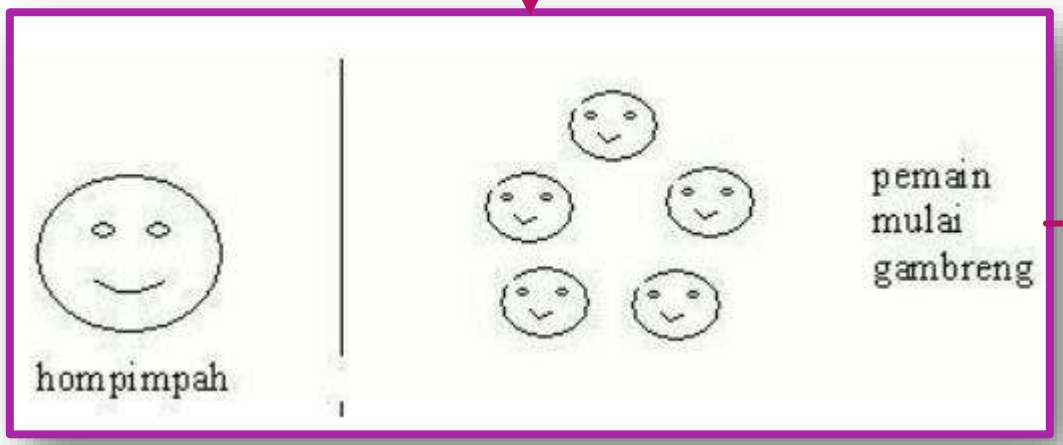
SEMAPHORE → Ilustrasi

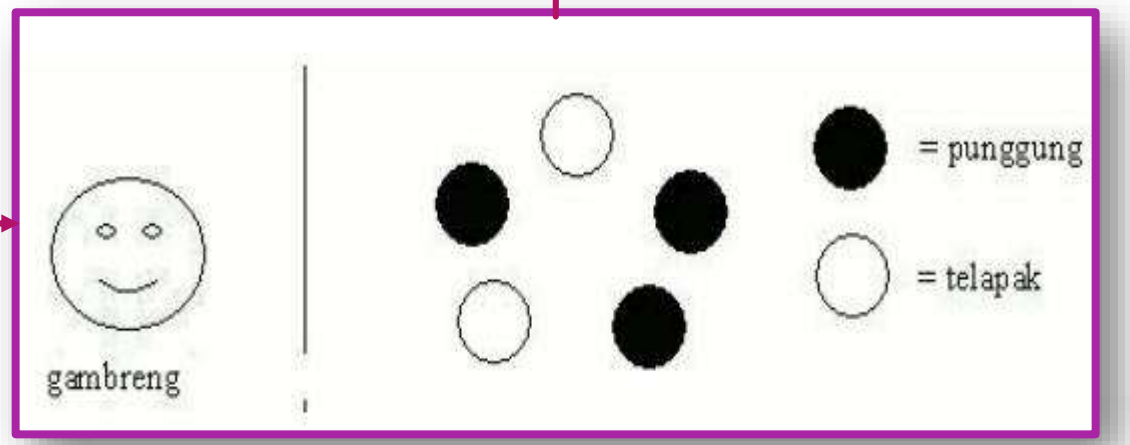


Semaphore

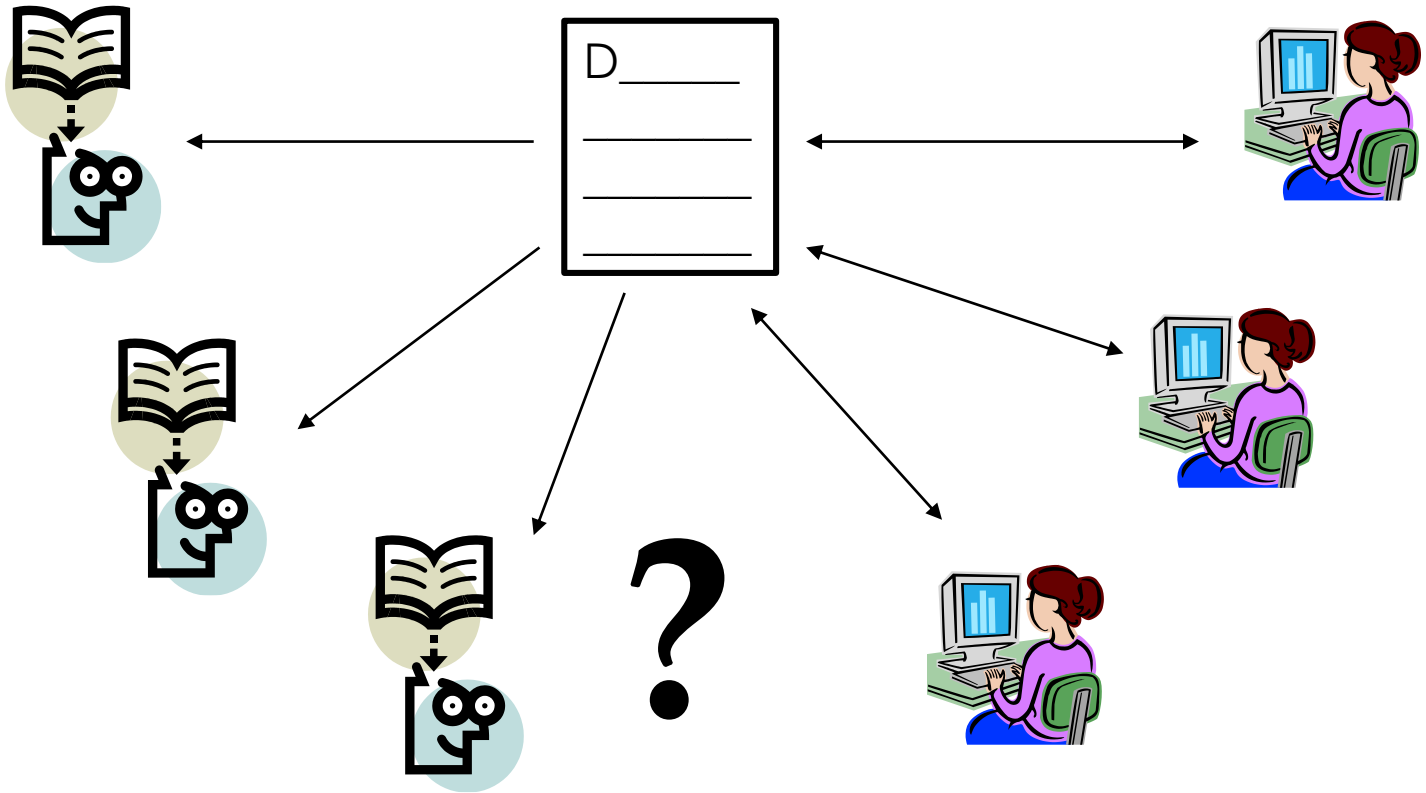
Thread







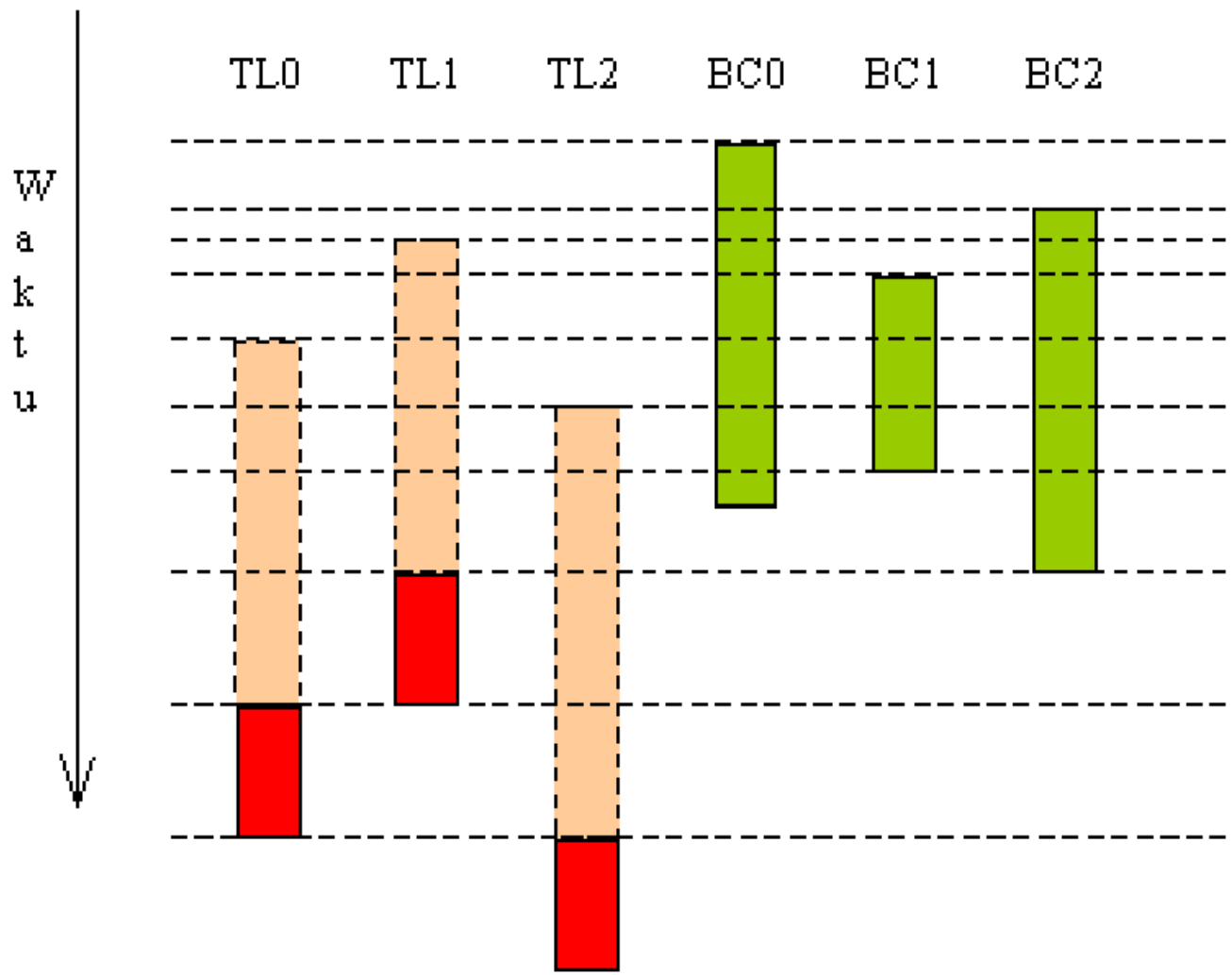
SEMAPHORE → Readers/Writers Problems



SEMAPHORE → Readers/Writers Problems

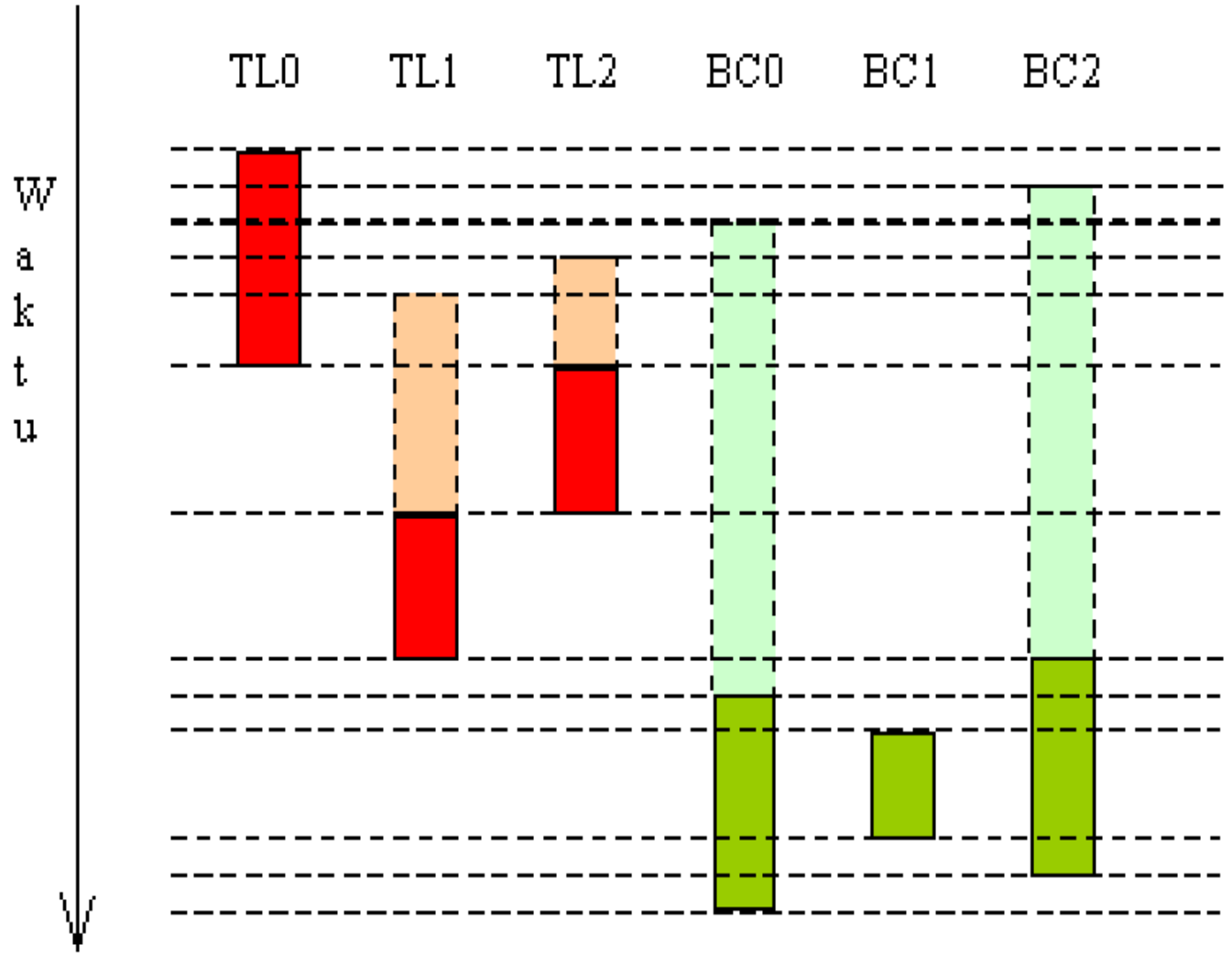
- Syarat :
  - Reader menjadi prioritas.
  - Writer menjadi prioritas.
  - Reader dan Writer memiliki prioritas yang sama.

Reader diprioritaskan



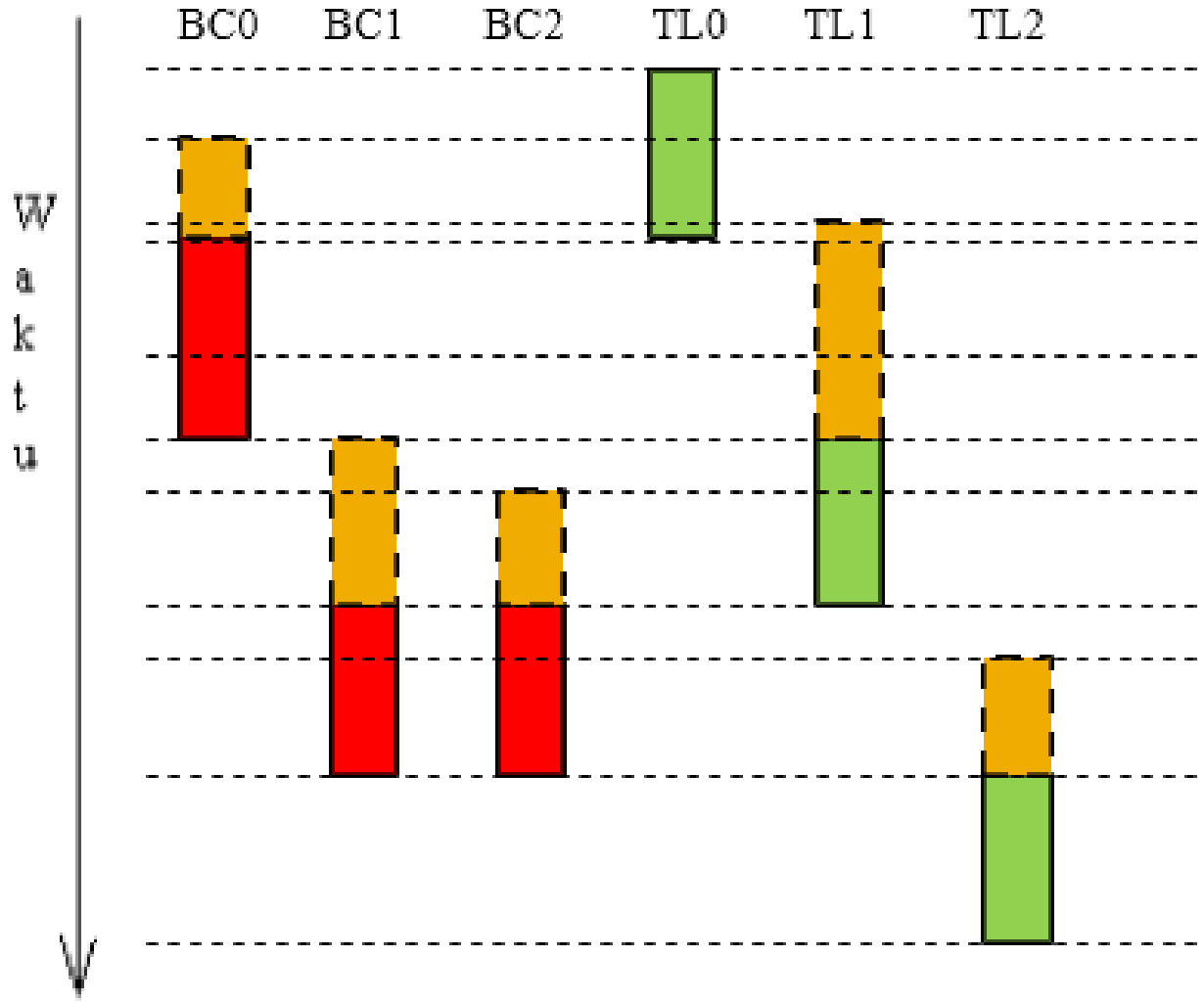
TL = Thread Penulis
BC = Thread Pembaca
= Thread menunggu
= Critical Section

Writers diprioritaskan



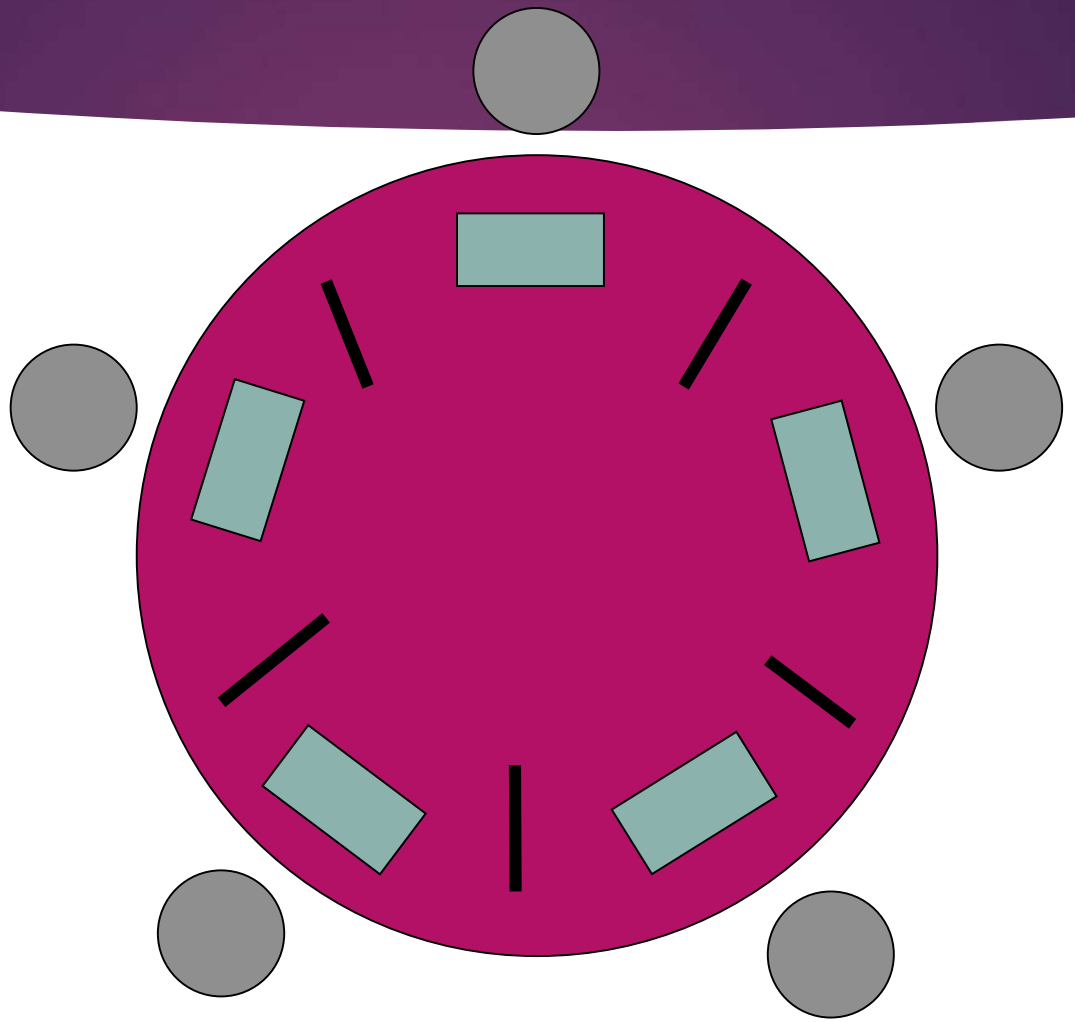
TL = Thread Penulis
BC = Thread Pembaca
= Thread menunggu
= Critical Section

Reader dan Writers diprioritaskan

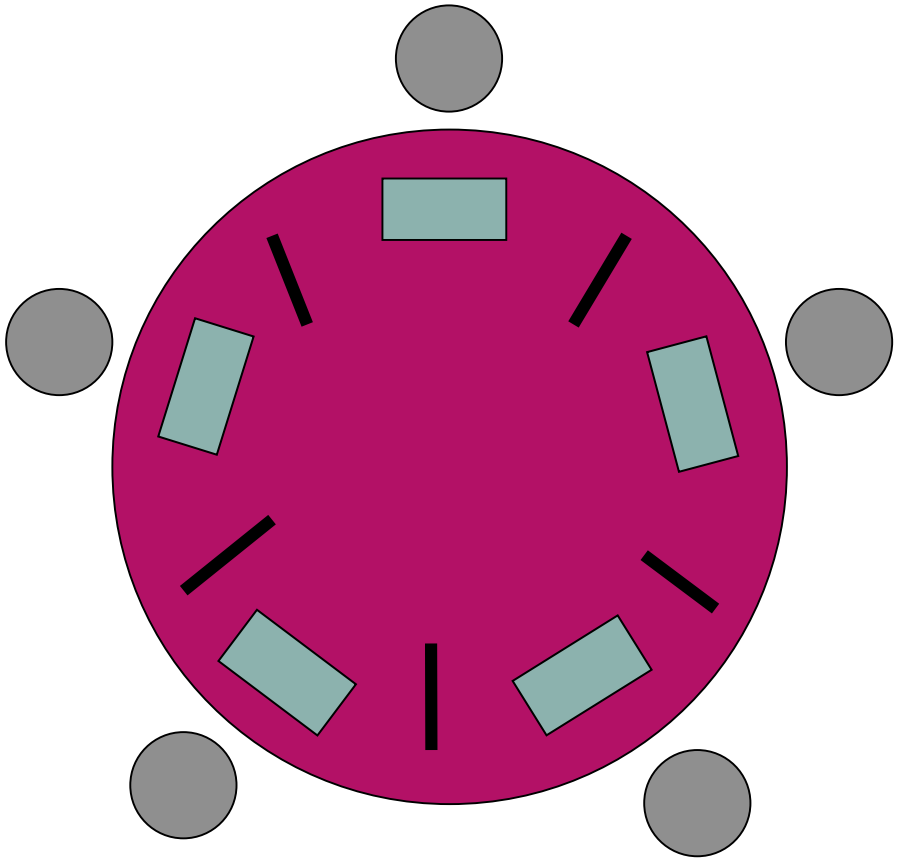


TL = Thread Penulis
BC = Thread Pembaca
= Thread menunggu
= Critical Section

SEMAPHORE → Dining Philosophers Problem

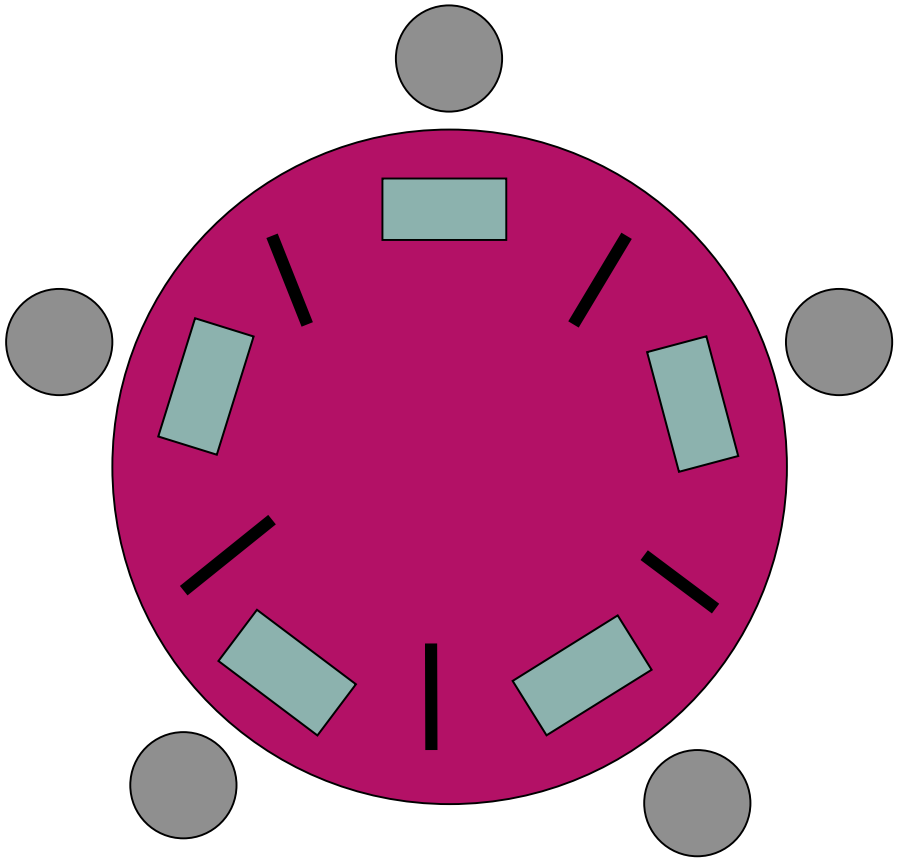


SEMAPHORE → Dining Philosophers Problem



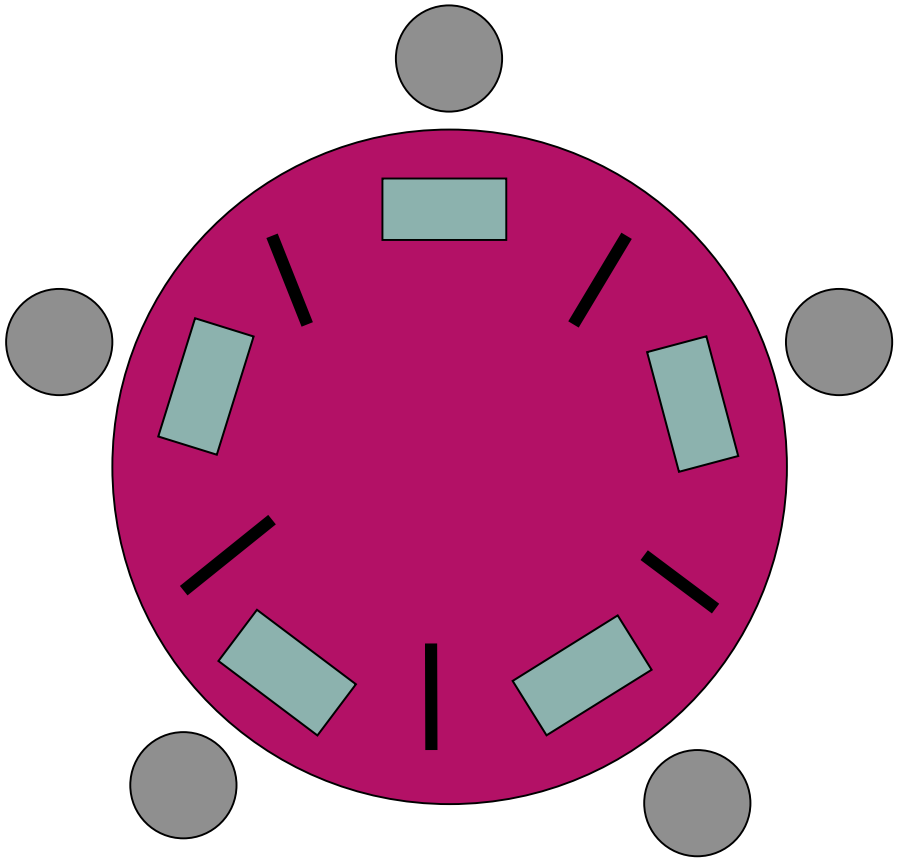
- Salah satu solusi yang mungkin langsung terlihat adalah dengan menggunakan semaphore.
- Setiap sumpit mewakili sebuah semaphore.
- Kemudian, ketika seorang filsuf lapar, maka dia akan mencoba mengambil sumpit di kiri dan di kanannya, atau dengan kata lain dia akan menunggu sampai kedua sumpit tersebut dapat ia gunakan.
- Setelah selesai makan, sumpit diletakkan kembali dan sinyal diberikan ke semaphore sehingga filsuf lain yang membutuhkan dapat menggunakan sumpitnya

SEMAPHORE → Dining Philosophers Problem



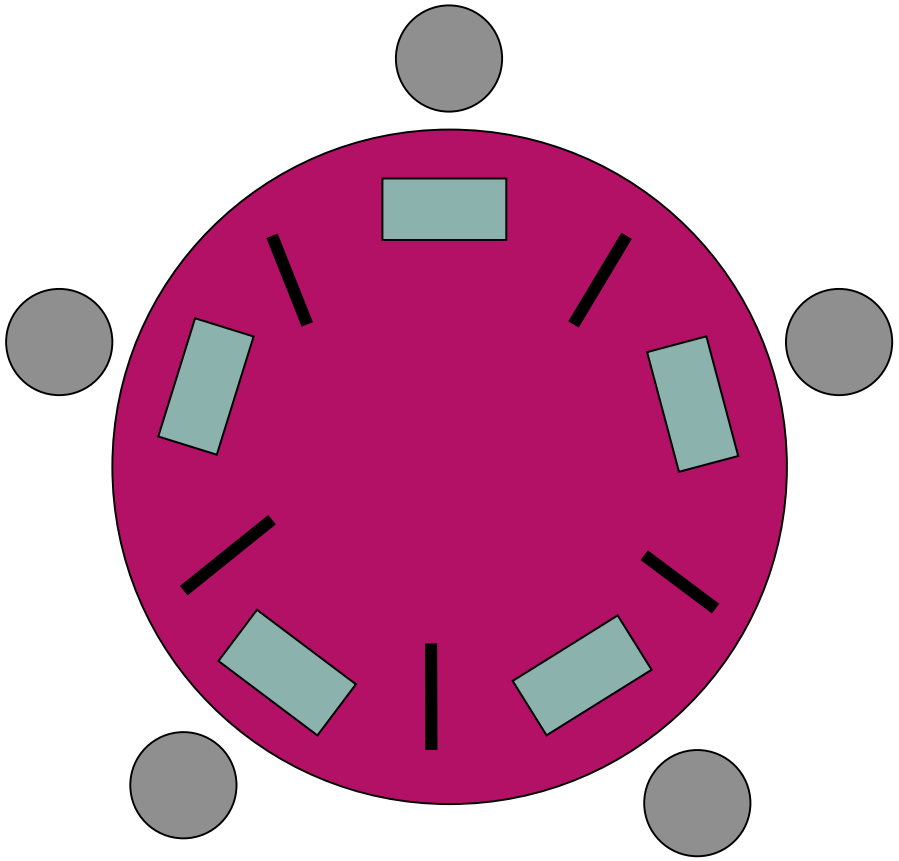
- Solusi-A, yaitu mengizinkan paling banyak 4 orang filsuf yang duduk bersama-sama pada satu meja. Tidak akan terjadi kondisi deadlock apabila terdapat kurang dari 4 orang filsuf yang duduk bersama-sama mengelilingi meja dengan 5 tempat duduk (satu filsuf dianggap mati).

SEMAPHORE → Dining Philosophers Problem



- Solusi-B, yaitu mengizinkan seorang filsuf mengambil sumpit hanya jika kedua sumpit itu ada. Apabila semua filsuf lapar secara bersamaan, maka hanya 2 orang filsuf yang dapat makan, karena filsuf mengambil 2 sumpit pada saat yang bersamaan.

SEMAPHORE → Dining Philosophers Problem



- Solusi-C, yaitu solusi asimetrik. Filsuf pada nomor ganjil mengambil sumpit kiri dulu baru sumpit kanan, sedangkan filsuf pada nomor genap mengambil sumpit kanan dulu baru sumpit kiri. Apabila semua filsuf lapar secara bersamaan, cara pengambilan dengan solusi asimetrik akan mencegah semua filsuf mengambil sumpit kiri secara bersamaan, sehingga kondisi deadlock dapat dihindari.

MONITOR → Ilusrasi