Sinkronisasi Proses

Semaphore

- Solusi permasalahan critical section yang dijelaskan pada bab sebelumnya sulit dilakukan untuk permasalahan yang lebih kompleks
  - Untuk mengatasinya digunakan alat sinkronisasi yang disebut "semaphore"
- Semaphore S merupakan variabel bertipe integer yang diakses dengan 2 operasi atomic standart, yaitu wait dan signal
- Operasi diwakili P (wait) dan V (signal) sbb

  wait(S) : while $S \leq 0$ do no-op;
  S:=S-1;

  signal(S) : S:=S+1;
  - Modifikasi nilai integer pada semaphore dalam operasi wait dan signal harus dieksekusi individual. Jika satu proses mengubah niai semaphore, tidak ada proses lain yang mengubah nilai semaphore yang sama
  - Pada wait(S), testing nilai integer S ($S \leq 0$) dan modifikasi yang mungkin (S:=S-1), harus juga dieksekusi tanpa interupsi

Kegunaan Semaphore

- Solusi permasalahan critical section untuk n proses
  - n proses membagi semaphore, mutex (mutual exclusion), diinisialisasi 1. Setiap proses P_i diorganisasikan sebagai berikut :

```
REPEAT
        wait(mutex);
                critical section
        signal(mutex);
                remainder section
UNTIL false
```

- Penggunaan semaphore untuk sinkronisasi dua proses yang dijalankan konkuren :
  - P1 dengan pernyataan S1 dan
  - P2 dengan pernyataan S2

```
/* pada proses P1 :
        S1;
        signal(synch);
/* pada proses P2 :
        wait(synch);
        S2;
```

  - Semaphore *synch* diinisialisasi 0

Implementasi Semaphore (1)

- Dilakukan modifikasi pada operasi wait dan signal.
- Jika proses mengeksekusi operasi wait, maka nilai semaphore menjadi tidak positif, pada saat itu proses memblok dirinya sendiri dan terjadi waiting queue
- Proses yang sedang diblok akan menunggu hingga semaphore S direstart, yaitu pada saat beberapa proses yang lain mengeksekusi operasi signal. Suatu proses akan direstart dengan operasi wakeup dan akan mengubah proses dari keadaan waiting ke ready

Implementasi Semaphore (2)

```
type semaphore = record
                    value: integer;
                    L: list of process;
                 end;
var S: semaphore;
wait(S) : S.value := S.value-1;
          if S.value < 0 then
               begin
                    tambahkan proses ini ke S.L
                    block;
               end;
signal(S) : S.value := S.value-1;
          if S.value ≤ 0 then
               begin
                    hapus proses dari S.L
                    wakeup(P);
               end;
```

variabel

operasi

Implementasi Semaphore (3)

- Akan tetapi implementasi diatas menyebabkan situasi deadlock, contohnya :

(inisial nilai S = Q = 1)

P0	P1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

Semaphore Biner

- Semaphore yang diterangkan sebelumnya disebut semaphore "counting", karena nilai integer dapat dijangkau sampai nilai tak hingga
- Semaphore "biner" adalah semaphore dengan nilai integer yang dapat dijangkau hanya antara 0 dan 1
- Implementasi S sebagai semaphore counting :
  - Struktur Data

```
VAR S1, S2, S3 : binary-semaphore;
        C : integer;
(nilai inisial S1=S3=1, S2=0, C diset nilai inisial semaphore
 counting S}
```

Semaphore Biner

- Operasi “wait” pada semaphore counting S :

```
wait(S3);
wait(S1);
C:=C-1;
IF C < 0 THEN
BEGIN
   signal(S1);
   wait(S2);
END
ELSE signal(S1);
signal(S3);
```

- Operasi “signal” pada semaphore counting S :

```
Wait(S1);
C:=C+1;
IF C <= 0 THEN signal(S2);
signal(S1);
```

Masalah Klasik Dalam Sinkronisasi

- Sejumlah permasalahan klasik sinkronisasi digunakan untuk testing skema sinkronisasi
- Dalam hal ini, kita gunakan semaphore untuk sinkronisasi permasalahan
- Terdiri dari :
  - The Bounded-Buffer (Producer-Consumer) Problem
  - The Reader and Writer Problem
  - The Dining-Philosophers Problem

The Bounder-Buffer Problem

- Diasumsikan pool berisi n buffer yang masing-masing digunakan menyimpan satu item
- Semaphore umum :
  - *mutex* (diinisialisasi 1} digunakan untuk mutual exclusion saat mengakses buffer
  - Semaphore empty menghitung jumlah buffer kosong (diinisialisasi n)
  - Semaphore full menghitung jumlah buffer penuh {diinisialisasi 1}

Semaphore mutex untuk Proses Producer dan Consumer

Struktur proses producer

```
REPEAT
    ....
    menghasilkan item di nextp
    ....
    wait(empty);
    wait(mutex);
    ...
    menambahkan nextp ke buffer
    ...
    signal(mutex);
    signal(full);
UNTIL false
```

Struktur proses consumer

```
REPEAT
    wait(full);
    wait(mutex);
    ....
    memindahkan item dari buffer ke
    nextc
    ....
    signal(mutex);
    signal(empty);
    ...
    mengkonsumsi item dari nextc
    ...
UNTIL false
```

The Reader and Writer Problem

- Obyek data (seperti file atau record) digunakan bersama-sama diantara beberapa proses yang konkuren. Beberapa proses mungkin ingin hanya membaca isi shared object, dan lainnya ingin mengubah (read dan write) shared object
- Ada 2 variasi masalah ini yaitu :
  - Seorang reader tidak perlu menunggu reader lain untuk selesai hanya karena ada writer menunggu (reader mempunyai prioritas lebih tinggi dari pada writer)
  - Jika ada writer yang sedang menunggu, maka tidak boleh ada reader lain yang bekerja (writer memiliki prioritas yang lebih tinggi)

Semaphore mutex untuk Proses Writer dan Reader

Variabel umum :

VAR mutex, wrt : semaphore; {diinisialisasi 1}
readcount : integer; {diinisialisasi 0}

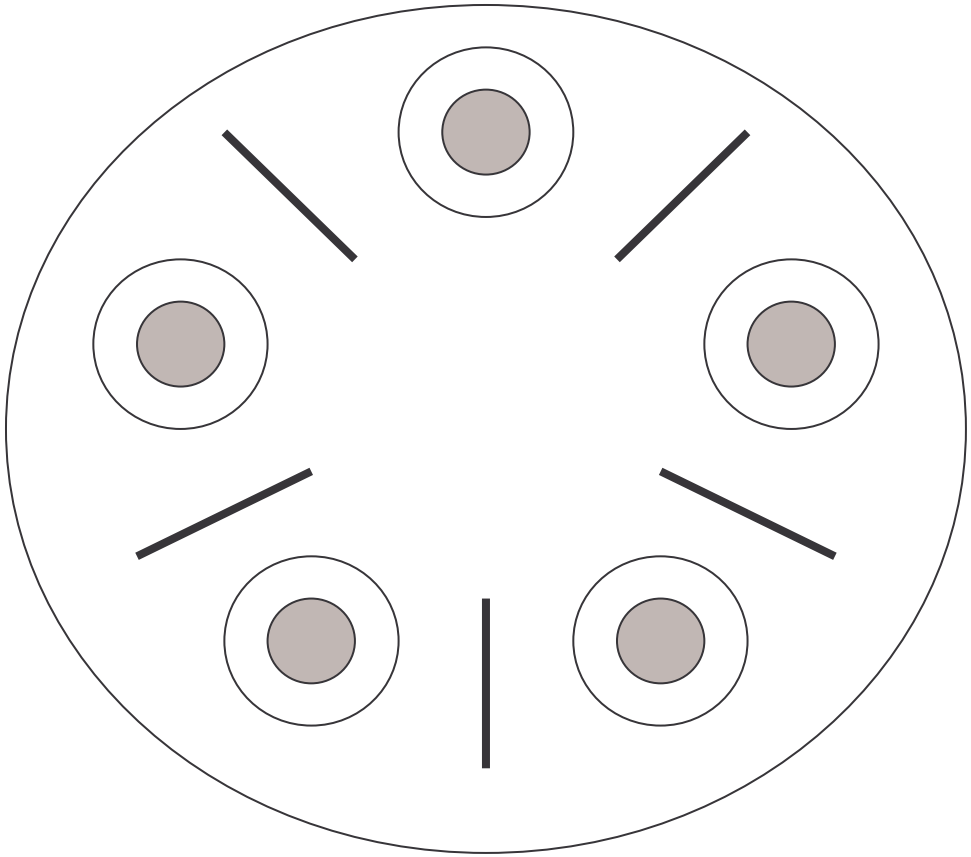
Proses writer	**Proses reader**

wait(wrt);
…
menulis
…
signal(wrt);
	wait(mutex);
	readcount := readcount+1;
	IF readcount = 1 THEN wait(wrt);
	signal(mutex);
	….
	membaca
	….
	wait(mutex);
	readcount := readcount-1;
	IF readcount = 0 THEN signal(wrt);
	signal(mutex);

Dining-Philosophers Problem (1)

- Diketahui 5 filosof menghabiskan hidupnya untuk berfikir dan makan
- Filosof tsb membagi meja melingkar dengan 5 kursi yang dimiliki setiap filosof
- Di tengah meja tersedia semangkuk nasi dan terdapat 5 supit
- Bila filosof berpikir, maka tidak berinteraksi dg tetangganya
- Bila filosof lapar akan mengambil 2 supit terdekat (sebelah kanan dan kirinya)
- filosof tidak dapat mengambil supit tetangganya yang sedang digunakan, harus menunggu tetangganya selesai menggunakan

Dining-Philosophers Problem (2)



Semaphore Chopstick untuk Solusi Dining-Philosophers Problem (1)

Struktur data :

VAR chopstick : ARRAY [0..4] of semaphore

{ diinisialisasi 1 }

```
REPEAT
    wait(chopstick[i]);
    wait(chopstick[i+5 mod 5]);
      ....
    makan
    ....
    signal(chopstick[i]);
    signal(chopstick[i+5 mod 5]);
    ...
    berfikir
    ...
UNTIL false
```

Semaphore Chopstick untuk Solusi Dining-Philosophers Problem (2)

- Solusi diatas menjamin tidak ada 2 tetangga yang makan bersama-sama, tapi masih memungkinkan terjadi deadlock
- Deadlock terjadi apabila semua filosof lapar dan mengambil supit kiri, maka semua nilai supit=0 dan sebaliknya
- Cara menghindari deadlock :
  - Mengijinkan paling banyak 4 orang filosof yang duduk bersama-sama pada satu meja
  - Mengijinkan seorang filosof mengambil supit hanya jika kedua supit itu tersedia
  - Menggunakan solusi asimetrik, yaitu filosof pada nomor ganjil mengambil supit kiri dulu baru supit kanan. Sedangkan filosof yang duduk di kursi genap mengambil supit kanan dulu baru supit kiri