Bab 5

Sinkronisasi Proses

POKOK BAHASAN:

- ✓ Permasalahan Critical Section
- ✓ Sinkronisasi Perangkat Keras
- ✓ Semaphore
- ✓ Masalah-masalah Klasik dalam Sinkronisasi

TUJUAN BELAJAR:

Setelah mempelajari materi dalam bab ini, mahasiswa diharapkan mampu:

- ✓ Memahami permasalahan critical section
- ✓ Memahami algoritma sinkronisasi
- ✓ Memahami konsep semaphore untuk sinkronisasi proses
- ✓ Memahami implementasi sinkronisasi dengan masalah-masalah klasik dalam sinkronisasi proses

5.1 LATAR BELAKANG

Proses-proses yang konkuren adalah proses-proses (lebih dari satu) berada pada saat yang sama. Proses-proses ini dapat sepenuhnya tak bergantung dengan yang lainnya, tapi dapat juga saling berinteraksi. Proses-proses yang berinteraksi memerlukan sinkronisasi agar terkendali dengan baik.

Proses-proses yang melakukan akses secara konkuren pada data yang digunakan bersama-sama menyebabkan data tidak konsisten (inconsistence). Agar data konsisten dibutuhkan mekanisme untuk menjamin eksekusi yang berurutan pada proses-proses yang bekerja sama. Pada model shared memory untuk penyelesaian

permasalahan bounded-buffer paling banyak menyimpan *n* – 1 item pada buffer pada saat yang bersamaan. Untuk mendapatkan solusi dimana semua *N* buffer digunakan bukan masalah yang sederhana. Misalnya dilakukan modifikasi kode producer-consumer dengan menambahkan variabel *counter* yang diinisialisasi 0 dan dinaikkan setiap satu item baru ditambahkan ke buffer. Definisi data yang digunakan bersama-sama (*shared data*) adalah sebagai berikut :

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

Proses pada *producer* akan menambahkan satu nilai variabel *counter* sebagai berikut :

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

Sebaliknya juga proses pada *consumer* akan menurunkan satu nilai variabel *counter* sebagai berikut :

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

Pernyataan **counter++** dan **counter--** harus dilakukan secara atomik. Operasi atomik adalah operasi yang harus menyelesaikan seluruh pernyataannya tanpa interupsi. Pernyataan **counter++** akan diimplementasikan dalam bahasa mesin sebagai berikut :

```
register1 = counter
register1 = register1 + 1
counter = register1
```

Sedangkan pernyataan **counter--** akan diimplementasikan dalam bahasa mesin sebagai berikut :

```
register1 = counter
register1 = register1 + 1
counter = register1
```

Apabila baik producer dan consumer mencoba mengubah buffer secara konkuren, pernyataan dalam bahasa mesin diatas akan dilakukan secara terpisah. Misalnya **counter** diinisialisasi 5. Pernyataan yang dijalankan adalah :

producer: **register1 = counter** (*register1 = 5*)

producer: **register1 = register1 + 1** (*register1 = 6*)

consumer: **register2 = counter** (*register2 = 5*)

consumer: **register2 = register2 – 1** (*register2 = 4*)

producer: **counter = register1** (*counter = 6*)

consumer: **counter = register2** (*counter = 4*)

Nilai **counter** kemungkinan bernilai 4 atau 6, sedangkan hasil yang benar seharusnya 5, hal ini yang dimaksud dengan data yang tidak konsisten. Situasi dimana beberapa proses mengakses dan memanipulasi data yang digunakan bersama-sama secara konkuren disebut dengan *race condition*. Nilai akhir dari data yang digunakan bersama-sama tersebut tergantung dari proses yang terakhir selesai. Untuk mencegah *race condition* tersebut, proses yang konkuren harus dilakukan sinkronisasi.

5.2 PERMASALAHAN CRITICAL-SECTION (CRITICAL-SECTION PROBLEM)

Suatu system terdiri dari *n* proses dimana semuanya berkompetisi menggunakan data yang digunakan bersama-sama. Masing-masing proses mempunyai sebuah kode segmen yang disebut dengan *critical section*, dimana proses memungkinkan untuk mengubah variabel umum, mengubah sebuah tabel, menulis file dan lain sebagainya. Gambaran penting dari sistem adalah, ketika sebuah proses dijalankan di dalam *critical section*, tidak ada proses lain yang diijinkan untuk menjalankan *critical section*-nya. Sehingga eksekusi dari *critical section* oleh proses-proses tersebut berlaku eksklusif (*mutually exclusive*). Permasalahan *critical section* digunakan untuk mendesain sebuah protokol dimana proses-proses dapat bekerja sama. Masing-masing proses harus meminta ijin untuk memasuki *critical section*-nya. Daerah kode yang mengimplementasikan perintah ini disebut daerah *entry*. *Critical section* biasanya diikuti oleh daerah *exit*. Kode pengingat terletak di daerah *remainder*.

Sebuah solusi dari permasalahan *critical section* harus memenuhi 3 syarat sebagai berikut :

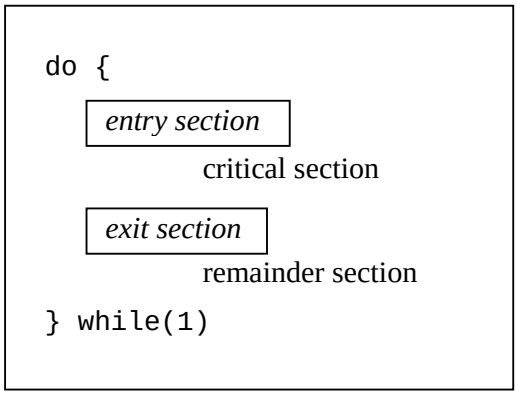
1. **Mutual Exclusion.** Apabila proses P_i menjalankan *critical section*-nya, maka tidak ada proses lain yang dapat menjalankan *critical section*.
2. **Progress.** Apabila tidak ada proses yang menjalankan *critical section*-nya dan terdapat beberapa proses yang akan memasuki *critical section*-nya, maka hanya proses-proses itu yang tidak diproses di dalam daerah pengingat (*remainder*) dapat ikut berpartisipasi di dalam keputusan proses mana yang akan memasuki *critical section* selanjutnya, dan pemilihan ini tidak dapat ditunda tiba-tiba.
3. **Bounded Waiting.** Terdapat batasan jumlah waktu yang diijinkan oleh proses lain untuk memasuki *critical section* setelah sebuah proses membuat permintaan untuk memasuki *critical section*-nya dan sebelum permintaan dikabulkan.

Asumsi bahwa masing-masing proses dijalankan pada kecepatan bukan nol (nonzero). Akan tetapi tidak ada asumsi mengenai kecepatan relatif dari proses ke *n*.

Pemecahan masalah *critical section* tidak mengandalkan semua asumsi tentang instruksi hardware atau jumlah prosessor dari hardware yang mendukung. Akan tetapi, diasumsikan bahwa instruksi dasar bahasa mesin (instruksi primitif, misalnya *load*, *store* dan *test*) dijalankan secara otomatis. Sehingga apabila dua instruksi dijalankan

bersama-sama, hasilnya ekuivalen dengan deret eksekusi dalam beberapa pesanan yang tidak diketahui. Sehingga apabila perintah *load* dan *store* dijalankan bersama-sama, perintah *load* akan mempunyai nilai lama atau nilai baru, tetapi tidak kombinasi dari kedua perintah itu.

Ketika mengimplementasikan suatu algoritma, kita menentukan dahulu hanya variable-variabel yang digunakan untuk keperluan sinkronisasi dan menggambarkan hanya proses P_i seperti struktur seperti Gambar 5-1. *Entry section* dan *exit section* di dalam kotak untuk menunjukkan segmen kode yang penting. Proses-proses kemungkinan menggunakan variabel-variabel umum untuk sinkronisasi kegiatannya.



Gambar 5-1 : Struktur umum dari proses P_i

5.2.1 Pemecahan Dua Proses

Pada sub bab ini kita membatasi pada algoritma yang dapat diaplikasikan hanya terhadap dua proses pada satu waktu. Proses tersebut diberi nama P_0 dan P_1. Untuk jelasnya, ketika menyatakan P_i, kita gunakan P_j untuk menyatakan proses yang lain, dimana $j = 1 - i$.

5.2.1.1 Algoritma 1

Pendekatan pertama adalah memperbolehkan semua proses menggunakan variable integer `turn` diinisialisasi ke 0 (atau 1).

```
int turn;
```

Apabila `turn = i`, maka proses P_i diijinkan untuk menjalankan *critical section* – nya. Struktur dari proses P_i adalah sebagai berikut :

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

Pemecahan ini menjamin hanya satu proses pada satu waktu yang dapat berada di critical section. Tetapi hal ini tidak memuaskan kebutuhan progress, karena hal ini membutuhkan proses lain yang tepat pada eksekusi dari *critical section*. Sebagai contoh, apabila `turn=0` dan P_1 siap untuk memasuki *critical section*, P_1 tidak dapat melakukannya, meskipun P_0 mungkin di dalam *remainder section* – nya.

5.2.1.2 Algoritma 2

Kelemahan dengan algoritma 1 adalah tidak adanya informasi yang cukup tentang *state* dari masing-masing proses. Untuk mengatasi masalah ini dilakukan penggantian variabel `turn` dengan array

```
boolean flag[2];
```

Inisialisasi awal `flag [0] = flag [1] = false`. Apabila `flag[i]` bernilai *true*, nilai ini menandakan bahwa P_i siap untukmemasuki *critical section*. Struktur dari proses P_i adalah sebagai berikut :

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

Pemecahan ini menjamin *mutual exclusion*, tetapi masih belum memenuhi *progress* .

5.2.1.3 Algoritma 3

Algoritma ini merupakan kombinasi algoritma 1 dan algoritma 2. Harapannya akan didapatkan solusi yang benar untuk masalah *critical-section*, dimana proses ini menggunakan dua variabel :

```
int turn;
boolean flag[2];
```

Inisialisasi `flagI[0] = flag[1] = false` dan nilai dari `turn` bernilai 0 atau 1. Struktur dari proses P_i adalah :

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

Algoritma ketiga ini memenuhi ketiga kebutuhan diatas yaitu *mutual exclusion*, *progress* dan *bounded waiting* dan memecahkan permasalahan *critical section* untuk dua proses.

5.2.2 Algoritma Bakery

Algoritma Bakery adalah algoritma yang digunakan untuk pemecahan permasalahan *critical section* pada *n* proses. Sebelum memasuki *critical section*, proses menerima nomo. Proses yang mempunyai nomor terkecil dapat memasuki *critical section*. Jika proses P_i dan P_j menerima nomor yang sama, jika $i < j$ maka P_i dilayani lebih dahulu, sebaliknya P_j akan dilayani lebih dahulu. Skema pemberian nomor selalu membangkitkan nomor dengan menaikkan nilai urut misalnya 1, 2, 3, 3, 3, 3, 4, 5, ….. Pada algoritma bakery terdapat notasi <≡ untuk urutan nomor (ticket #, process id #) sebagai berikut :

- $(a,b) < (c,d)$ if $a < c$ or if $a = c$ and $b < d$

- max (*a0*,…, *an*-1) is a number, *k*, such that $k \geq ai$ for i - 0, …, $n - 1$

Variabel umum yang digunakan adalah :

```
boolean choosing[n];
int number[n];
```

Struktur data diatas diinisialisasi `false` dan `0`. Struktur dari proses P_i adalah :

```
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);
```

5.3 PERANGKAT KERAS SINKRONISASI

Pada sistem multiprosessor, proses-pmroses bertindak independen. Interupsi di satu pemroses tidak mempengaruhi pemroses-pemroses yang lain. Pemroses-pemroses yang memakai memori bersama maka pengaksesan terhadap suatu memori dijaga pada tingkat perangkat keras agar tidak boleh pemroses lain tidak dapat mengakses suatu lokasi yang sama di saat yang sama.

Perancang perangkat keras menyediakan instruksi-instruksi atomik yang tak dapat diinterupsi. Instruksi dilaksanakan sampai selesai. Instruksi ini biasanya dilaksanakan dengan cara mngunci bus sehingga pemroses-pemroses lain tak dapat menggunakan bus. Pemroses yang mengunci bus dengan leluasa membaca dan/atau memodifikasi suatu lokasi memori.

Beragam instruksi mesin disediakan oleh perancang pemroses guna membantu implementasi *mutual exclusion*. Diantara instruksi-instruksi itu adalah:

- *tsl* (*test and set lock*)
- *tas* atau *ts* (*test and set*), digunakan IBM S/360, keluarga Motorola M68000, dan lain-lain
- *cs* (*compare and set*), digunakan IBM 370 series
- *exchange* (*xchg*), digunakan intel X86
- dsb

5.3.1 Metode Test and Set

Metode Test and Set melakukan testing dan memodifikasi isi memori secara atomik menggunakan fungsi Test and Set sebagai berikut :

```
boolean TestAndSet (boolean &target)
{
        boolean rv = target;
        tqrget = true;
        return rv;
}
```

Untuk menyelesaikan permasalahan *mutual exclusion* dengan metode Test and Set maka digunakan variable umum berikut :

```
boolean lock = false;
```

Sedangkan Process P_i mempunyai struktur sebagai berikut :

```
do {
      while (TestAndSet(lock)) ;
            critical section
      lock = false;
            remainder section
}
```

5.3.2 Metode Swap

Metode swap menggunakan prosedur swap untuk menukar dua variable secara atomic. Prosedur swap adalah sebagai berikut :

```
void Swap(boolean &a, boolean &b) {
      boolean temp = a;
      a = b;
      b = temp;
}
```

Untuk menyelesaikan permasalahan *mutual exclusion* menggunakan prosedur swap, variabel umum yang digunakan adalah

```
boolean lock;
boolean waiting[n];
```

Kedua variable diatas diinisialisasi `false`. Sedangkan struktur process P_i adalah sebagai berikut :

```
do {
      key = true;
      while (key == true)
            Swap(lock,key);
                  critical section
      lock = false;
            remainder section
}
```

5.4 SEMAPHORE

Semaphore adalah pendekatan yang dikemukakan Dijkstra. Prinsip semaphore adalah 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 strukstur penanda yang sesuai kebutuhannya. Variabel khusus untuk penandaan ini disebut *semaphore*.

Semaphore adalah alat untuk sinkronisasi yang tidak membutuhkan *busy waiting*. *Semaphore S* berupa variable integer. *Semaphore* hanya dapat diakses melalui operasi atomic yang tak dapat diinterupsi sampai kode selesai. Operasi dari *semaphore S* adalah *wait* dan *signal* berikut :

wait (*S*):

while *S*≤ 0 do *no-op*;

***S*--;**

signal (*S*):

***S*++;**

Adanya *semaphore* mempermudah penyelesaian persoalan *critical section* pada *n* proses. Penyelesaian *critical section* menggunakan *semaphore* menggunakan variabel umum berikut :

```
semaphore mutex;
```

Variabel *semaphore* `mutex` diinisialisasi `mutex = 1`. Sedangkan struktur program untuk proses P_i adalah :

```
do {
     wait(mutex);
          critical section
     signal(mutex);
          remainder section
} while (1);
```

Implementasi *semaphore* harus dapat menjamin *mutual exclusion* variabel *semaphore*, yaitu hanya mengijinkan satu proses pada satu saat yang boleh memanipulasi *semaphore*. Implementasi sebuah *semaphore* menggunakan struktur data record sebagai berikut :

```
typedef struct {
        int value;
        struct process *L;
} semaphore;
```

Pada *semaphore* terdapat dua operasi sederhana yaitu `block` untuk menghentikan sementara proses yang menggunakan *semaphore* dan `wakeup(P)` untuk melanjutkan eksekusi proses *P* yang di-blok. Operasi *wait* dan *signal* dari *semaphore* didefinisikan sebagai :

wait(*S*):

```
S.value--;
if (S.value < 0)  {
```

```
        tambahkan proses ke S.L;
        block;
    }
```

signal(*S*):

```
    S.value++;
    if (S.value <= 0) {
        hapus proses P dari S.L;
        wakeup(P);
    }
```

Sebagai alat sinkronisasi yang umum, *semaphore* dieksekusi oleh suatu proses setelah proses lain. Misalnya *semaphore B* pada proses P_j hanya dieksekusi setelah *semaphore A* dieksekusi pada proses P_i. Pada saat menggunakan *semaphore*, *flag* diinisialisasi 0. Kode yang diakses proses P_i dan P_j dapat dilihat berikut ini :

P_i	P_j
⋮	⋮
A	*wait*(*flag*)
signal(*flag*)	*B*

Semaphore merupakan salah satu sumber daya sistem. Misalnya dua proses P_1 dan P_2, dua sumber daya kritis R_1 dan R_2, proses P_1 dan P_2 harus mengakses kedua sumber daya. Kondisi berikut dapat terjadi : R_1 diberikan ke P_1, sedang R_2 diberikan ke P_2. Apabila dua proses untuk melanjutkan eksekusi memerlukan kedua sumber daya sekaligus maka kedua proses akan saling menunggu sumber daya lain selamanya. Tak ada proses yang dapat melepaskan sumber daya yang telah dipegangnya karena menunggu sumber daya lain yang tak pernah diperolehnya. Kedua proses dalam kondisi *deadlock*, tidak dapat membuat kemajuan apapun.

Pada saat beberapa proses membawa *semaphore* dan masing-masing proses menunggu *semaphore* yang sedang dibawa oleh proses lain maka kemungkinan akan terjadi *deadlock*. Misalnya terdapat dua *semaphore S* dan *Q* yang diinisialisasi 1.

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal(*S*);	*signal*(*Q*);
signal(*Q*);	*signal*(*S*);

Proses P_0 dan P_1 masing-masing menjalankan operasi *wait(S)* dan *wait(Q).* Kemudian proses P_0 dan P_1 menjalankan operasi *wait(Q)* dan *wait(S)* maka sistem akan *deadlock* sampai salah satu proses menjalankan operasi *signal*.

Apabila suatu proses tidak pernah dihapus dari antrian *semaphore* setelah suatu *semaphore* dihentikan sementara, maka terjadi bloking yang tak terbatas. Keadaan ini disebut *starvation.*

Keadaan starvation digambarkan sebagai berikut. Misalnya terdapat tiga proses P_1, P_2 dan P_3 yang memerlukan pengaksesan sumber daya *R* secara periodik. Skenario yang bisa terjadi :

- P_1 sedang diberi sumber daya *R*, P_2 dan P_3 *blocked* menunggu sumber daya *R*.
- Ketika P_1 keluar dari *critical section*, P_2 dan P_3 diijinkan mengakses *R*.
- Asumsi P_3 diberi hak akses. Kemudian setelah selesai, hak akses kembali diberikan ke P_1 yang saat itu kembali membutuhkan sumber daya *R*.

Jika pemberian hak akses bergantian terus-menerus antara P_1 dan P_3, maka P_2 tidak pernah memperoleh pengaksesan sumber daya *R*, meski tidak ada *deadlock*. Pada situasi ini, P_2 mengalami yang disebut *Starvation.*

Terdapat dua bentuk *semaphore* yaitu *counting semaphore* dan *binary semaphore*. *Counting semaphore* menggunakan nilai integer yang mempunyai jangkauan tak terbatas seperti pada struktur yang telah dijelanskan diatas. *Binary semaphore* menggunakan nilai integer dengan jangkauan antara 0 dan 1 sehingga implementasinya lebih sederhana. *Counting semaphore S* diatas dapat diimplementasikan dengan *binary semaphore.* Struktur data yang digunakan adalah :

```
binary-semaphore S1, S2;
int C:
```

Struktur data diatas diinisialisasi dengan

```
S1 = 1
S2 = 0
```

C = initial value of semaphore S

Implementasi operasi *wait* dan *signal* pada *binary semaphore S* adalah sebagai berikut :

Operasi *wait* :

```
wait(S1);
C--;
if (C < 0) {
     signal(S1);
     wait(S2);
}
signal(S1);
```

Operasi *signal* :

```
wait(S1);
C ++;
if (C <= 0)
     signal(S2);
else
     signal(S1);
```

5.5 MASALAH-MASALAH KLASIK SINKRONISASI

Untuk mengimplementasikan permasalahan sinkronisasi dapat menggunakan model yang digunakan untuk permasalahan *Bounded Buffer, Reader Writer* dan *Dining Philosopher* yang akan dijelaskan di bawah ini.

5.5.1 Bounded-Buffer (Producer-Consumer) Problem

Produsen menghasilkan barang dan konsumen yang akan menggunakannya. Ada beberapa batasan yang harus dipenuhi, antara lain :

- Barang yang dihasilkan oleh produsen terbatas
- Barang yang dipakai konsumen terbatas

- Konsumen hanya boleh menggunakan barang yang dimaksud setelah produsen menghasilkan barang dalam jumlah tertentu
- Produsen hanya boleh memproduksi barang jika konsumen sudah kehabisan barang

Untuk penyelesaian permasalahan *bounded buffer* menggunakan *semaphore* menggunakan variabel umum berikut :

```
semaphore full, empty, mutex;
```

Inisialisasi untuk variable diatas, `full = 0`, `empty = n`, `mutex = 1.` Struktur program untuk produsen adalah

```
do {
        ...
        menghasilkan item pada nextp
        ...
    wait(empty);
    wait(mutex);
        ...
        menambah nextp ke buffer
        ...
    signal(mutex);
    signal(full);
} while (1);
```

Sedangkan struktur program untuk konsumen adalah

```
do {
    wait(full)
    wait(mutex);
        ...
        mengambil item dari buffer ke nextc
        ...
    signal(mutex);
    signal(empty);
        ...
        menggunakan item pada nextc
        ...
} while (1);
```

5.5.2 Reader and Writer Problem

Terdapat dua variasi pada masalah ini, yaitu :

a. seorang *reader* tidak perlu menuggu *reader* lain untuk selesai hanya karena ada *writer* menunggu (*reader* memiliki prioritas lebih tinggi disbanding dengan *writer*)
b. Jika ada *writer* yang sedang menunggu, maka tidak boleh ada *reader* lain yang bekerja (*writer* memiliki prioritas yang lebih tinggi)

Jika terdapat *writer* dalam *critical section* dan terdapat `n` *reader* yang menunggu, maka satu *reader* akan antri di `wrt` dan `n-1` *reader* akan antri di `mutex`. Jika *writer* mengeksekusi `signal(wrt)`, maka dapat disimpulkan bahwa eksekusi adalah menunggu *reader* atau menunggu satu *writer*. Variabel umum yang digunakan adalah

```
semaphore mutex, wrt;
```

Inisialisasi variable diatas adalah `mutex = 1, wrt = 1, readcount = 0`. Struktur proses *writer* adalah

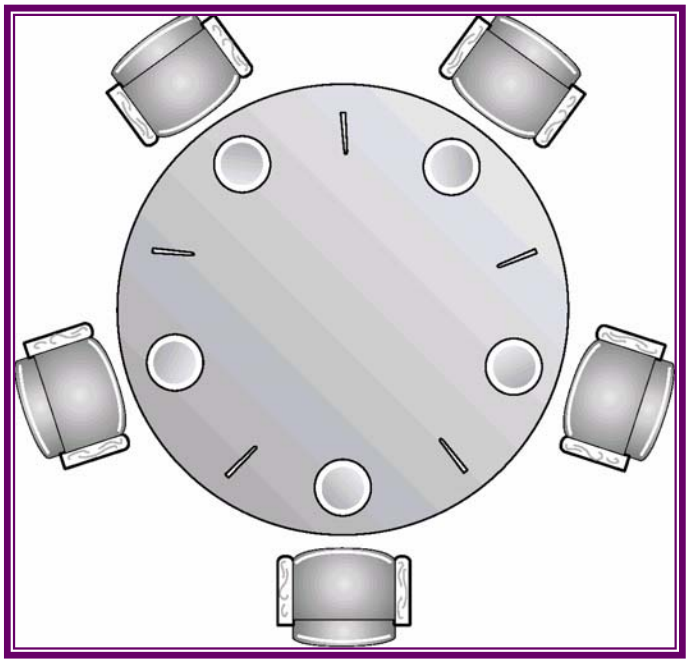
```
wait(wrt);
      ...
      menulis
      ...
signal(wrt);
```

Sedangkan struktur proses *reader* adalah

```
wait(mutex);
readcount++;
if (readcount == 1)
     wait(rt);
signal(mutex);
      ...
      membaca
      ...
wait(mutex);
readcount--;
if (readcount == 0)
     signal(wrt);
signal(mutex):
```

5.5.3 Dining-Philosophers Problem

Permasalahan *dining-philosophers* digambarkan pada Gambar 5-2 dimana terdapat 5 filosof yang akan makan. Di sana disediakan 5 supit. Jika filosof lapar, ia akan mengambil 2 supit yaitu di tangan kanan dan kiri. Namun adakalanya hanya diambil supit satu saja. Jika ada filosof yang mengambil 2 supit, maka ada filosof yang harus menunggu sampai supit tersebut diletakkan. Hal ini dapat diimplementasikan dengan *wait* dan *signal*.



Gambar 5-2 : Lima filosof dalam satu meja makan

Struktur data yang digunakan untuk penyelesaian permasalahan ini dengan *semaphore* adalah

```
semaphore chopstick[5];
```

Dimana semua nilai array dinisialisasi 1. Struktur program untuk filosof ke *i* adalah

```
do {
      wait(chopstick[i])
      wait(chopstick[(i+1) % 5])
            …
            makan
            …
```

```
            signal(chopstick[i]);
            signal(chopstick[(i+1) % 5]);
                  ...
                  berfikir
                  ...
      } while (1);
```

Meskipun solusi ini menjamin bahwa tidak ada 2 tetangga yang makan bersama-sama, namun masih mungkin terjadi deadlock, yaitu jika tiap-tiap filosof lapar dan mengambil supit kiri, maka semua nilai supit = 0, dan juka kemudian tiap-tiap filosof akan mengambil supit kanan, maka akan terjadi *deadlock*. Ada beberapa cara untuk menghindari *deadlock*, antara lain :

a. mengijinkan paling banyak 4 orang filosof yang duduk bersama-sama pada satu meja.
b. Mengijinkan seorang filosof mangambil supit hanya jika kedua supit itu ada (dengan catatan, bahwa ia harus mengambil pada critical section)
c. Menggunakan suatu solusi asimetrik, yaitu filosof pada nomor ganjil mengambil supit kanan dulu baru supit kiri. Sedangkan filosof yang duduk di kursi genap mengambil supit kanan dulu baru supit kiri.

5.6 CONTOH SINKRONISASI

5.6.1 Sinkronisasi pada Solaris 2

Pada Solaris 2, sinkronisasi diimplementasikan dengan menggunakan beberapa kunci untuk mendukung sistem multitasking, multithreading (termasuk thread real time) dan multiprosessing. Solaris 2 menggunakan *adaptive mutex* untuk efisiensi sistem pada saat proteksi data dari kode segment yang pendek. Selain itu juga menggunakan variabel kondisi dan kunci *reader writer* apabila kode segmen lebih panjang memerlukan akses ke data. Solaris 2 juga menggunakan *turnstile* untuk mengurutkan daftar thread yang menunggu untuk memperoleh baik *adaptive mutex* atau kunci *reader writer*.

5.6.2 Sinkronisasi pada Windows 2000

Implementasi sinkronisasi pada Windows 2000 menggunakan *interrupt mask* untuk memproteksi akses ke sumber daya global pada sistem uniprosessor sedangkan ada sistem multiprosessor menggunakan *spinlock*. Selain itu Windows 2000 juga menyediakan *dispatcher object* yang berfungsi sebagai *mutual exclusion* dan *semaphore*. *Dispatcher object* juga menyediakan *event* yang berfungsi sebagai variabel kondisi.

RINGKASAN:

LATIHAN SOAL :

1. Apa yang dimaksud dengan race condition?
2. Apakah yang dimaksud dengan critical section ? Untuk menyelesaikan masalah critical section , ada tiga hal yang harus dipenuhi, sebutkan dan jelaskan !
3. Bagaimana algoritma Bakery untuk sinkronisasi banyak proses (n proses) ?
4. Apa yang dimaksud semaphore dan sebutkan operasi pada semaphore
5. Bagaimana struktur semaphore yang digunakan untuk menyelesaikan permasalahan :
    a. *bounded buffer problem.*
    b. *reader and writer problem*.
    c. *dining philosopher problem*.