LABORATORIUM JARINGAN KOMPUTER
FAKULTAS ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA

PRAKTIKUM SISTEM OPERASI

SEMESTER : GENAP **TAHUN : 2015/2016**

BAB III

JUDUL BAB : SYNCHRONIZATION

DISUSUN OLEH : MOH ARIF ANDRIAN

NIM : 156150600111002

ASISTEN : SISKA PERMATASARI
ZAENAL KURNIAWAN

KOORDINATOR ASISTEN : DANY RAHMANA

LAPORAN PRAKTIKUM SISTEM OPERASI
FAKULTAS ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA

Nama : Moh. Arif Andrian
NIM : 156150600111002
Laporan : BAB III
Asisten : Siska Permatasari
Zaenal Kurniawan

BAB III
SYNCHRONIZATION

3.1 DASAR TEORI

3.1.1 Thread Synchronization

Thread library menyediakan tiga cara untuk mekanisme sinkronisasi, yaitu:

- Mutex – Mutual Exclusion lock: digunakan untuk mem-blokir akses suatu variabel dari thread lain. Hal ini memastikan bahwa suatu variabel dalam suatu waktu hanya bisa dimodifikasi oleh suatu thread tertentu saja
- Join – Membuat suatu thread hanya berjalan saat ada thread lain yang sudah selesai (terminate)
- Condition Variable – menggunakan tipe data *pthread_cond_t*

3.1.2 Mutex

Mutex digunakan untuk menghindari ketidakkonsistenan data karena sebuah variabel yang dimodifikasi oleh banyak thread dalam waktu yang bersamaan. Hal ini menyebabkan terjadinya apa yang disebut dengan *race condition*. Oleh karena itu, setiap ada variabel global yang bisa dimodifikasi oleh banyak thread, digunakan mutex untuk menjaga agar variabel global tersebut dimodifikasi oleh banyak threads dengan suatu urutan tertentu. Mutex berfungsi seperti lock and key, di mana jika suatu thread memodifikasi suatu variabel global, maka variabel tersebut hanya bisa dimodifikasi oleh thread lainnya setelah thread sebelumnya selesai memodifikasi variabel tersebut. Mutex ini hanya bisa digunakan oleh thread-thread yang berasal dari satu proses yang sama, tidak seperti semaphore yang dapat digunakan untuk banyak thread dari proses yang berbeda-beda.

- *Pthread_mutex_lock()* – digunakan untuk mengunci suatu variabel. Jika mutex sudah di-lock oleh suatu thread, maka thread lain tidak bisa memanggil fungsi ini sampai mutex tersebut sudah di-unlock
- *Pthread_mutex_unlock()* – digunakan untuk membuka kunci suatu variabel. Pesan error akan muncul jika suatu thread memanggil fungsi ini padahal bukan thread tersebut yang mengunci mutex
- *Pthread_mutex_trylock()* – digunakan untuk mencoba mengunci suatu mutex dan akan mengembalikan pesan error jika terjadi keadaan *busy*. Hal ini berguna untuk mencegah terjadinya deadlock

3.1.3 Join

Sebuah join digunakan jika diinginkan suatu keadaan sebuah thread baru akan berjalan setelah suatu thread lain selesai dijalankan. Sebuah thread dapat juga menjalankan suatu rutin yang menciptakan banyak thread baru di mana thread induk akan dijalankan lagi saat banyak thread baru tersebut sudah selesai dijalankan.

- *Pthread_create()* – membuat sebuah thread baru
- *Pthread_join()* – menunggu termination dari suatu thread lain
- *Pthread_self()* – mengembalikan nilai identifier dari thread yang sekarang dijalankan

3.1.4 Condition Variable

Sebuah condition variable adalah variabel dari *pthread_cond_t* dan digunakan sebagai sebuah fungsi untuk menunggu dan melanjutkan proses eksekusi suatu thread. Menggunakan perintah ini dapat membuat sebuah thread berhenti dijalankan dan berjalan lagi berdasarkan suatu kondisi yang memenuhinya.

- *Creating/Destroying: pthread_cond_init; pthread_cond_t cond=PTHREAD_COND_INITILIZER; pthread_cond_destroy*
- *Waiting on condition: pthread_cond_wait; pthread_cond_timewait*
- *Waking thread based on condition: pthread_cond_signal; pthread_cond_broadcast*

3.2 MATERI PRAKTIKUM

1. Login ke sistem GNU/Linux kemudian buka terminal.

```
andrian@156150600111002:~$ sudo su
[sudo] password for andrian:
root@156150600111002:/home/andrian#
```

2. Lakukan percobaan terhadap coding-coding berikut.
3. Compile menggunakan **gcc –pthread main.c –o main** lalu run menggunakan **./main**
4. Amati hasil running-nya untuk tiap percobaan dan buat kesimpulannya

3.2.1 Mutex

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <pthread.h>
4
5   void *functionC();
6   pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
7   int  counter = 0;
8
9   main()
10  {
11     int rc1, rc2;
12     pthread_t thread1, thread2;
```

```
13
14    /* Create independent threads each of which will
15 execute functionC */
16
17    if( (rc1=pthread_create( &thread1, NULL, &functionC,
18 NULL)) )
19    {
20       printf("Thread creation failed: %d\n", rc1);
21    }
22
23    if( (rc2=pthread_create( &thread2, NULL, &functionC,
24 NULL)) )
25    {
26       printf("Thread creation failed: %d\n", rc2);
27    }
28
29    /* Wait till threads are complete before main
30 continues. Unless we  */
31    /* wait we run the risk of executing an exit which
32 will terminate   */
33    /* the process and all threads before the threads
34 have completed.   */
35
36    pthread_join( thread1, NULL);
37    pthread_join( thread2, NULL);
38
39    exit(EXIT_SUCCESS);
40 }
41
42 void *functionC()
   {
      pthread_mutex_lock( &mutex1 );
      counter++;
      printf("Counter value: %d\n",counter);
      pthread_mutex_unlock( &mutex1 );
   }
```

PERCOBAAN:

1. Pertama buat file dengan isi sebagai berikut :

```
  GNU nano 2.5.3                    File: mutex.c

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
   int rc1, rc2;
   pthread_t thread1, thread2;

   /* Create independent threads each of which will execute functionC */

   if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
   {
      printf("Thread creation failed: %d\n", rc1);
   }

   if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
   {
      printf("Thread creation failed: %d\n", rc2);
   }

   /* Wait till threads are complete before main continues. Unless we  */
   /* wait we run the risk of executing an exit which will terminate   */
   /* the process and all threads before the threads have completed.   */

   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);
```

2. Compile file dengan perintah seperti pada gambar :

```
root@156150600111002:/home/andrian/Sinkronisasi# gcc -pthread -o mutex mutex.c
mutex.c:9:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 main()
 ^
```

3. Jalankan program dengan perintah sebagai berikut :

```
root@156150600111002:/home/andrian/Sinkronisasi# ./mutex
Counter value: 1
Counter value: 2
root@156150600111002:/home/andrian/Sinkronisasi#
```

Kesimpulan :

Pada kasus ini, mutex dapat mencegah terjadinya race condition dan juga deadlock dengan lock and key. Saat thread 1 mengakses variable counter maka thread 2 tidak bisa mengaksesnya sampai thread 1 selesai mengakses variable counter itu. Berlaku juga sebaliknya

3.2.2 Join

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  #define NTHREADS 10
```

```
5   void *thread_function(void *);
6   pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
7   int  counter = 0;
8
9   main()
10  {
11     pthread_t thread_id[NTHREADS];
12     int i, j;
13
14     for(i=0; i < NTHREADS; i++)
15     {
16        pthread_create(       &thread_id[i],       NULL,
17  thread_function, NULL );
18     }
19
20     for(j=0; j < NTHREADS; j++)
21     {
22        pthread_join( thread_id[j], NULL);
23     }
24
25     /* Now that all threads are complete I can print the
26  final result.     */
27     /* Without the join I could be printing a value before
28  all the threads */
29     /*           have           been           completed.
30  */
31
32     printf("Final counter value: %d\n", counter);
33  }
34
35  void *thread_function(void *dummyPtr)
36  {
37     printf("Thread number %ld\n", pthread_self());
       pthread_mutex_lock( &mutex1 );
       counter++;
       pthread_mutex_unlock( &mutex1 );
    }
```

PERCOBAAN:

1. Pertama buat file dengan isi sebagai berikut :

```
  GNU nano 2.5.3                File: join.c

#include <stdio.h>
#include <pthread.h>

#define NTHREADS 10
void *thread_function(void *);
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
   pthread_t thread_id[NTHREADS];
   int i, j;

   for(i=0; i < NTHREADS; i++)
   {
      pthread_create( &thread_id[i], NULL, thread_function, NULL );
   }

   for(j=0; j < NTHREADS; j++)
   {
      pthread_join( thread_id[j], NULL);
   }

   /* Now that all threads are complete I can print the final result.     */
   /* Without the join I could be printing a value before all the threads */
   /* have been completed.                                                */

   printf("Final counter value: %d\n", counter);
}

void *thread_function(void *dummyPtr)
{
```

2. Compile file dengan perintah seperti pada gambar :

```
root@156150600111002:/home/andrian/Sinkronisasi# gcc -pthread -o join join.c
join.c:9:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 main()
 ^
```

3. Jalankan program dengan perintah sebagai berikut :

```
root@156150600111002:/home/andrian/Sinkronisasi# ./join
Thread number 140502089086720
Thread number 140502080694016
Thread number 140502072301312
Thread number 140502063908608
Thread number 140502055515904
Thread number 140502047123200
Thread number 140502038730496
Thread number 140502030337792
Thread number 140502021945088
Thread number 140502013552384
Final counter value: 10
root@156150600111002:/home/andrian/Sinkronisasi#
```

Kesimpulan :

Pada kasus ini, join dapat membuat thread dapat berjalan saat thread yang lain sudah terminate. Dengan kata lain fungsi join adalah mengatur thread untuk melakukan

proses secara bergantian. Ketika thread yang lain sedang dijalankan maka thread yang lain akan menunggu dan akan dijalankan ketika thread yang lain terminate.

3.2.3 Condition Variable

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  pthread_mutex_t     count_mutex                    =
5  PTHREAD_MUTEX_INITIALIZER;
6  pthread_cond_t           condition_var             =
7  PTHREAD_COND_INITIALIZER;
8
9  void *functionCount1();
10 void *functionCount2();
11 int  count = 0;
12 #define COUNT_DONE  10
13 #define COUNT_HALT1  3
14 #define COUNT_HALT2  6
15
16 main()
17 {
18    pthread_t thread1, thread2;
19
20    pthread_create(  &thread1,  NULL,  &functionCount1,
21 NULL);
22    pthread_create(  &thread2,  NULL,  &functionCount2,
23 NULL);
24
25    pthread_join( thread1, NULL);
26    pthread_join( thread2, NULL);
27
28    printf("Final count: %d\n",count);
29
30    exit(EXIT_SUCCESS);
31 }
32
33 //  Write  numbers  1-3  and  8-10  as  permitted  by
34 functionCount2()
35
36 void *functionCount1()
37 {
38    for(;;)
39    {
40       // Lock mutex and then wait for signal to relase
41 mutex
42       pthread_mutex_lock( &count_mutex );
```

```
43
44        // Wait while functionCount2() operates on count
45        //  mutex  unlocked  if  condition  varialbe  in
46  functionCount2() signaled.
47        pthread_cond_wait(                &condition_var,
48  &count_mutex );
49        count++;
50        printf("Counter                                value
51  functionCount1: %d\n",count);
52
53        pthread_mutex_unlock( &count_mutex );
54
55        if(count >= COUNT_DONE) return(NULL);
56      }
57  }
58
59  // Write numbers 4-7
60
61  void *functionCount2()
62  {
63      for(;;)
64      {
65         pthread_mutex_lock( &count_mutex );
66
67         if( count < COUNT_HALT1 || count > COUNT_HALT2 )
68         {
69            // Condition of if statement has been met.
70            // Signal  to  free  waiting  thread  by  freeing
71  the mutex.
72            // Note: functionCount1() is now permitted to
73  modify "count".
74            pthread_cond_signal( &condition_var );
75         }
76         else
77         {
            count++;
            printf("Counter                                value
    functionCount2: %d\n",count);
         }

         pthread_mutex_unlock( &count_mutex );

         if(count >= COUNT_DONE) return(NULL);
      }

    }
```

PERCOBAAN:

1. Pertama buat file dengan isi sebagai berikut :

```
  GNU nano 2.5.3                File: COnditionVariabel.c

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t count_mutex     = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condition_var   = PTHREAD_COND_INITIALIZER;

void *functionCount1();
void *functionCount2();
int  count = 0;
#define COUNT_DONE  10
#define COUNT_HALT1  3
#define COUNT_HALT2  6

main()
{
   pthread_t thread1, thread2;

   pthread_create( &thread1, NULL, &functionCount1, NULL);
   pthread_create( &thread2, NULL, &functionCount2, NULL);

   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);

   printf("Final count: %d\n",count);

   exit(EXIT_SUCCESS);
}

// Write numbers 1-3 and 8-10 as permitted by functionCount2()

void *functionCount1()
```

2. Compile file dengan perintah seperti pada gambar :

```
root@156150600111002:/home/andrian/Sinkronisasi# gcc -pthread -o COnditionVariabel COnditionVariabel.c
COnditionVariabel.c:15:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 main()
 ^
```

3. Jalankan program dengan perintah sebagai berikut :

```
root@156150600111002:/home/andrian/Sinkronisasi# ./COnditionVariabel
Counter value functionCount1: 1
Counter value functionCount1: 2
Counter value functionCount1: 3
Counter value functionCount2: 4
Counter value functionCount2: 5
Counter value functionCount2: 6
Counter value functionCount2: 7
Counter value functionCount1: 8
Counter value functionCount1: 9
Counter value functionCount1: 10
Final count: 10
root@156150600111002:/home/andrian/Sinkronisasi#
```

Kesimpulan :

Pada kasus diatas, conditional variable membuat thread 1 yang telah dieksekusi menunggu thread 2 yang akan dieksekusi selanjutnya. Sedangkan thread 1 akan kembali dieksekusi setelah thread 2 selesai dieksekusi.

LAPORAN PRAKTIKUM SISTEM OPERASI
FAKULTAS ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA

Nama : Moh. Arif Andrian
NIM : 156150600111002
Tugas : BAB III
Asisten : Siska Permatasari
Zaenal Kurniawan

TUGAS PRAKTIKUM

1. Jelaskan kenapa perlu sinkronisasi ?

   Jawaban:

   Menghindari ketidak konsistenan data karena sebuah variabel yang dimodifikasi oleh banyak thread dalam waktu yang bersamaan.

2. Jelaskan secara singkat penggunaan dari mutex, join, dan condition variable ?

   Jawaban:

   ***Mutex*:** jika suatu thread memodifikasi suatu variabel global, maka variabel tersebut hanya bisa dimodifikasi oleh thread lainnya setelah thread sebelumnya selesai memodifikasi variabel tersebut.

   ***Join*:** Digunakan jika diinginkan suatu keadaan sebuah thread baru akan berjalan setelah suatu thread lain selesai dijalankan. Sebuah thread dapat juga menjalankan suatu rutin yang menciptakan banyak thread baru di mana thread induk akan dijalankan lagi saat banyakthread baru tersebut sudah selesai dijalankan.

   ***Conditon variable*:** digunakan sebagai sebuah fungsi untuk menunggu dan melanjutkan proses eksekusi suatu thread. Menggunakan perintah ini dapat membuat sebuah thread berhenti dijalankan dan berjalan lagi berdasarkan suatu kondisi yang memenuhinya.

3. Perhatikan kode berikut ini

```
1   #include<stdio.h>
2   #include<string.h>
3   #include<pthread.h>
4   #include<stdlib.h>
5   #include<unistd.h>
6
7   pthread_t tid[2];
8   int counter;
9
10  void* doSomeThing(void *arg)
11  {
12      unsigned long i = 0;
```

```
13      counter += 1;
14      printf("\n Job %d started\n", counter);
15
16      for(i=0; i<(0xFFFFFFFF);i++);
17
18      printf("\n Job %d finished\n", counter);
19
20      return NULL;
21  }
22
23  int main(void)
24  {
25      int i = 0;
26      int err;
27
28      while(i < 2)
29      {
30          err    =    pthread_create(&(tid[i]),    NULL,
31  &doSomeThing, NULL);
32          if (err != 0)
33              printf("\ncan't   create   thread   :[%s]",
34  strerror(err));
35          i++;
36      }
37
38      pthread_join(tid[0], NULL);
39      pthread_join(tid[1], NULL);
40
        return 0;
    }
```

```
  GNU nano 2.5.3                File: tugas-4.1.c

#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>

pthread_t tid[2];
int counter;

void* doSomeThing(void *arg)
{
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d started\n", counter);

    for(i=0; i<(0xFFFFFFFF);i++);

    printf("\n Job %d finished\n", counter);

    return NULL;
}

int main(void)
{
    int i = 0;
    int err;
```

Kode tersebut akan memberikan keluaran

```
dataq@so:~/code$ gcc -pthread -o tugas-4.1 tugas-4.1.c
dataq@so:~/code$ ./tugas-4.1

 Job 1 started

 Job 2 started

 Job 2 finished

 Job 2 finished
dataq@so:~/code$
```

```
root@156150600111002:/home/andrian/Sinkronisasi# gcc -pthread -o tugas-4.1 tugas-4.1.c
root@156150600111002:/home/andrian/Sinkronisasi# ./tugas-4.1

 Job 1 started

 Job 2 started

 Job 2 finished

 Job 2 finished
root@156150600111002:/home/andrian/Sinkronisasi#
```

Jelaskan kenapa hal tersebut bisa terjadi !

Jawaban:

Secara umum program yang berisi thread tersebut dijalankan tanpa menggunakan metode sinkronisasi sehingga penggunaan variabel gloabal (counter) pada suatu waktu tidak dapat diatur. Sehingga pada saat thread pertama menggunakan variabel counter, thread kedua juga dapat menggunakan variabel counter yang menyebabkan thread kedua selesai terlebih dahulu.

4. Jika kode tersebut dimodifikasi sehingga terlihat seperti berikut ini

```
1   #include<stdio.h>
2   #include<string.h>
3   #include<pthread.h>
4   #include<stdlib.h>
5   #include<unistd.h>
6
7   pthread_t tid[2];
8   int counter;
9   pthread_mutex_t lock;
10
11  void* doSomeThing(void *arg)
12  {
13      pthread_mutex_lock(&lock);
14
15      unsigned long i = 0;
16      counter += 1;
17      printf("\n Job %d started\n", counter);
18
19      for(i=0; i<(0xFFFFFFFF);i++);
20
21      printf("\n Job %d finished\n", counter);
22
23      pthread_mutex_unlock(&lock);
24
25      return NULL;
26  }
27
28  int main(void)
29  {
30      int i = 0;
31      int err;
32
33      if (pthread_mutex_init(&lock, NULL) != 0)
34      {
```

```
35          printf("\n mutex init failed\n");
36          return 1;
37      }
38
39      while(i < 2)
40      {
41          err     =     pthread_create(&(tid[i]),     NULL,
42  &doSomeThing, NULL);
43          if (err != 0)
44              printf("\ncan't   create   thread   :[%s]",
45  strerror(err));
46          i++;
47      }
48
49      pthread_join(tid[0], NULL);
50      pthread_join(tid[1], NULL);
51      pthread_mutex_destroy(&lock);
52
        return 0;
}
```

```
 GNU nano 2.5.3                File: tugas-4.2.c

    for(i=0; i<(0xFFFFFFFF);i++);

    printf("\n Job %d finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
}

int main(void)
{
    int i = 0;
    int err;

    if (pthread_mutex_init(&lock, NULL) != 0)
    {
        printf("\n mutex init failed\n");
        return 1;
    }

    while(i < 2)
    {
        err = pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);
        if (err != 0)
            printf("\ncan't create thread :[%s]", strerror(err));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
```

Kode tersebut akan memberikan keluaran

```
dataq@so:~/code$ gcc -pthread -o tugas-4.2 tugas-4.2.c
dataq@so:~/code$ ./tugas-4.2

 Job 1 started

 Job 1 finished

 Job 2 started

 Job 2 finished
dataq@so:~/code$
```

```
root@156150600111002:/home/andrian/Sinkronisasi# gcc -pthread -o tugas-4.2 tugas-4.2.c
root@156150600111002:/home/andrian/Sinkronisasi# ./tugas-4.2

 Job 1 started

 Job 1 finished

 Job 2 started

 Job 2 finished
root@156150600111002:/home/andrian/Sinkronisasi#
```

Jelaskan kenapa hal tersebut bisa terjadi ! Bagaimana kaitannya dengan kode tersebut sebelum dimodifikasi ?

Jawaban:

Secara umum program yang berisi thread tersebut dijalankan dengan menggunakan metode sinkronisasi sehingga penggunaan variabel gloabal (counter) pada suatu waktu dapat diatur. Sehingga pada saat thread pertama menggunakan variabel counter, thread kedua tidak dapat menggunakan variabel counter yang menyebabkan thread pertama dijalankan dan selesai terlebih dahulu dibandingkan thread kedua.

LAPORAN PRAKTIKUM SISTEM OPERASI
FAKULTAS ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA

Nama : Moh. Arif Andrian
NIM : 156150600111002
Kesimpulan : BAB III
Asisten : Siska Permatasari
Zaenal Kurniawan

KESIMPULAN

Sinkronisasi merupakan suatu proses secara bersama sama dan saling berbagi data bersama dapat mengakibatkan race condition atau inkosistensi data. Sinkornisasi di perlukan untuk menghindari terjadinya ketidak konsistenan data akibat adanya akses secara konkuren. Proses-proses tersebut disebut konkukuren jika proses itu ada dan berjalan pada waktu yang bersamaan. Sinkronisasi diperlukan untuk menghindari ketidak konsistenan data karena sebuah variabel yang dimodifikasi oleh banyak thread dalam waktu yang bersamaan.

Thread library menyediakan tiga cara dalam mekanisme singkronisasi data, yaitu :

a. Mutex atau Mutual Exclusion Lock
   Jika suatu thread memodifikasi suatu variabel global, maka variabel tersebut hanya bisa dimodifikasi oleh thread lainnya setelah thread sebelumnya selesai memodifikasi variabel tersebut.
b. Join
   Digunakan jika diinginkan suatu keadaan sebuah thread baru akan berjalan setelah suatu thread lain selesai dijalankan. Sebuah thread dapat juga menjalankan suatu rutin yang menciptakan banyak thread baru di mana thread induk akan dijalankan lagi saat banyakthread baru tersebut sudah selesai dijalankan.
c. Condition Variabe
   Digunakan sebagai sebuah fungsi untuk menunggu dan melanjutkan proses eksekusi suatu thread. Menggunakan perintah ini dapat membuat sebuah thread berhenti dijalankan dan berjalan lagi berdasarkan suatu kondisi yang memenuhinya.