



Nama: **Zointa Ras Bangun (120140225)**
Mata Kuliah: **Sistem Operasi RD (IF2223)**

Tugas Ke: **2**
Tanggal: **11/04/2022**

1 Tujuan HandsOn

HandsOn 2 bertujuan untuk memahami - sistem sinkronisasi dan permasalahan yang ada, Memahami solusi dalam menangani *critical section*, Memahami implementasi dari: - 'join' menggunakan semaphores - Binary Semaphores - Producer Consumer - Reader / Writer Locks - Dining Philosophers

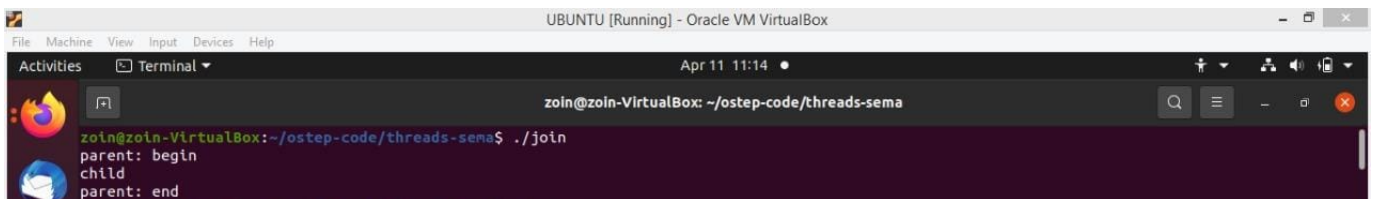
2 Fork/Join

2.1 Source Code

```
1  class SynthiaDataset(Dataset):
2      #include <stdio.h>
3      #include <stdlib.h>
4      #include <pthread.h>
5      #include <unistd.h>
6
7      #include "common.h"
8      #include "common_threads.h"
9
10     #ifdef linux
11     #include <semaphore.h>
12     #elif APPLE
13     #include "zemaphore.h"
14     #endif
15
16     sem_t s;
17
18     void *child(void *arg) {
19         sleep(2);
20         printf("child\n");
21         Sem_post(&s); // signal here: child is done
22         return NULL;
23     }
24
25     int main(int argc, char *argv[]) {
26         Sem_init(&s, 0);
27         printf("parent: begin\n");
28         pthread_t c;
29         Pthread_create(&c, NULL, child, NULL);
30         Sem_wait(&s); // wait here for child
31         printf("parent: end\n");
32         return 0;
33     }
34
```

2.2 Output

Berikut contoh output dari terminal yang saya buat :



```

zoin@zoin-VirtualBox: ~/ostep-code/threads-sema$ ./join
parent: begin
child
parent: end

```

Gambar 1: Tampilan *fork/join*

2.3 Penjelasan

Semaphore bisa dikatakan sebuah struktur data komputer yang berguna dalam sinkronisasi proses dan berfungsi dalam memerintsh jalannya proses suatu program. contohnya adalah suatu *thread* menunggu *list* supaya *list* tersebut berisi atau tidak kosong. Dari kondisi tersebut, semaphore tadi akan di definisikan dan di inisiasi menjadi 0 oleh Sem init. Maksud dari proses tersebut ialah semaphore akan dibagi antara threads pada proses yang sama. Kemudian apabila pembuatan thread sudah selesai akan dilanjutkan pemanggilan fungsi child semaphore yang akan melakukan sinyal bahwa proses child sudah selesai dan mulai me-return. Apabila child sudah selesai, maka semaphore akan melanjutkannya dan mengeluarkan output "parent : end".

3 Binary Semaphores

3.1 Source Code

```

1  class SynthiaDataset(Dataset):
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <pthread.h>
5  #include <unistd.h>
6
7  #include "common.h"
8  #include "common_threads.h"
9
10 #ifdef linux
11 #include <semaphore.h>
12 #elif APPLE
13 #include "zemaphore.h"
14 #endif
15
16 sem_t mutex;
17 volatile int counter = 0;
18
19 void *child(void *arg) {
20     int i;
21     for (i = 0; i < 10000000; i++) {
22         Sem_wait(&mutex);
23         counter++;
24         Sem_post(&mutex);
25     }
26     return NULL;
27 }
28
29 int main(int argc, char *argv[]) {
30     Sem_init(&mutex, 1);

```

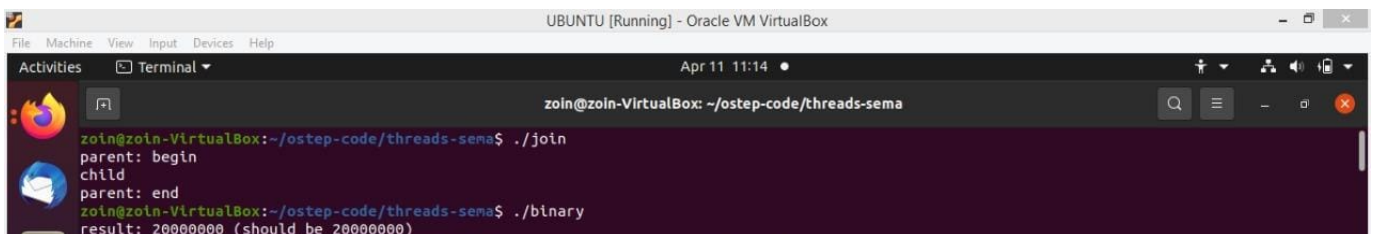
```

31 pthread_t c1, c2;
32 pthread_create(&c1, NULL, child, NULL);
33 pthread_create(&c2, NULL, child, NULL);
34 pthread_join(c1, NULL);
35 pthread_join(c2, NULL);
36 printf("result: %d (should be 20000000)\n", counter);
37 return 0;
38 }
39

```

3.2 Output

Berikut contoh output dari terminal yang saya buat :



Gambar 2: Tampilan *Binary*

3.3 Penjelasan *Binary Semaphores*

pada kode diatas mendefinisikan dan menginisialisasikan semaphore mutex dengan value sebesar 1. lalu dibuat lah thread yang berinisial c1 dan c2 berguna dalam menjalankan child. selanjutnya, akan dilakukan penginisialisasian pada perulangan sampai nilai tersebut kurang dari 1000000 yang mana akan menjalankan *sem wait* dan di saat itu juga *value* akan berkurang dan mulai dilakukan *critical section*. akan ada penambahan nilai counter yang kemudian semaphore memproses calling dengan menambah value dari semaphore

4 Producer/Consumer

4.1 Source Code

```

1
2 class SynthiaDataset(Dataset):
3     #include <stdio.h>
4     #include <unistd.h>
5     #include <assert.h>
6     #include <pthread.h>
7     #include <stdlib.h>
8
9     #include "common#include <stdio.h>
10 #include <unistd.h>
11 #include <assert.h>
12 #include <pthread.h>
13 #include <stdlib.h>
14
15 #include "common.h"
16 #include "common_threads.h"
17
18 #ifdef linux
19 #include <semaphore.h>
20 #elif __APPLE__

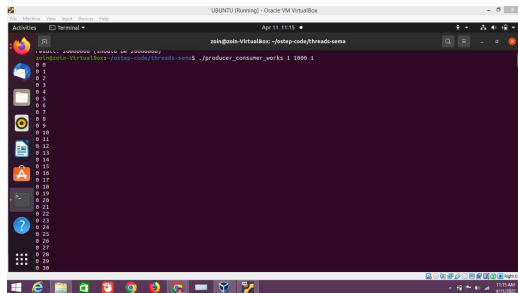
```

```
21 #include "zemaphore.h"
22 #endif
23
24 int max;
25 int loops;
26 int *buffer;
27
28 int use = 0;
29 int fill = 0;
30
31 sem_t empty;
32 sem_t full;
33 sem_t mutex;
34
35 #define CMAX (10)
36 int consumers = 1;
37
38 void do_fill(int value) {
39     buffer[fill] = value;
40     fill++;
41     if (fill == max)
42         fill = 0;
43 }
44
45 int do_get() {
46     int tmp = buffer[use];
47     use++;
48     if (use == max)
49         use = 0;
50     return tmp;
51 }
52
53 void *producer(void *arg) {
54     int i;
55     for (i = 0; i < loops; i++) {
56         Sem_wait(&empty);
57         Sem_wait(&mutex);
58         do_fill(i);
59         Sem_post(&mutex);
60         Sem_post(&full);
61     }
62
63     // end case
64     for (i = 0; i < consumers; i++) {
65         Sem_wait(&empty);
66         Sem_wait(&mutex);
67         do_fill(-1);
68         Sem_post(&mutex);
69         Sem_post(&full);
70     }
71
72     return NULL;
73 }
74
75 void *consumer(void *arg) {
76     int tmp = 0;
77     while (tmp != -1) {
78         Sem_wait(&full);
79         Sem_wait(&mutex);
80         tmp = do_get();
81         Sem_post(&mutex);
82         Sem_post(&empty);
```

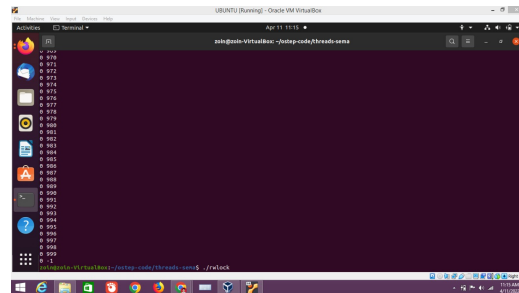
```
83 printf("%lld %d\n", (long long int) arg, tmp);
84 }
85 return NULL;
86 }
87
88 int main(int argc, char *argv[]) {
89     if (argc != 4) {
90         fprintf(stderr, "usage: %s <buffersize> <loops> <consumers>\n", argv[0]);
91         exit(1);
92     }
93     max = atoi(argv[1]);
94     loops = atoi(argv[2]);
95     consumers = atoi(argv[3]);
96     assert(consumers <= CMAX);
97
98     buffer = (int *) malloc(max * sizeof(int));
99     assert(buffer != NULL);
100     int i;
101     for (i = 0; i < max; i++) {
102         buffer[i] = 0;
103     }
104
105     Sem_init(&empty, max); // max are empty
106     Sem_init(&full, 0);    // 0 are full
107     Sem_init(&mutex, 1);   // mutex
108
109     pthread_t pid, cid[CMAX];
110     Pthread_create(&pid, NULL, producer, NULL);
111     for (i = 0; i < consumers; i++) {
112         Pthread_create(&cid[i], NULL, consumer, (void *) (long long int) i);
113     }
114     Pthread_join(pid, NULL);
115     for (i = 0; i < consumers; i++) {
116         Pthread_join(cid[i], NULL);
117     }
118     return 0;
119 }
120
```

4.2 Output

Berikut contoh output dari terminal yang saya buat :



(a) Augment Result 1



(b) Augment Result 2

Gambar 3: Screenshot Percobaan

4.3 Penjelasan Producer Consumer

Implementasi Producer/Consumer disebut bounded buffer. Isi program tersebut ialah memanggil, mengurangi, menghalangi konsumen, dan menunggu thread lain agar dapat memanggil Sem post saat terjadi full. Selanjutnya, program akan memulai fungsi procedure yang berguna dalam memanggil Sem wait(empty) dan Sem post(mutex). Pada fungsi prosedur juga menjalankan terus sampai empty tadi menjadi max. Producer akan melakukan pengisian dengan fungsi do fill di entry pertama buffer setelah empty berkurang hingga mencapai nilai 0. Berikutnya, producer akan terus berjalan sampai suatu saat nanti memanggil Sem post(mutex) dan Sem post(full) yang mana akan mengganti nilai value full dari nilai -1 menjadi 0. Sehingga, Consumer akan melakukan fungsi looping ulang dan memblok dengan value empty semaphore bernilai kosong.

5 Reader or Writer Locks

5.1 Source Code

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5
6 #include "common.h"
7 #include "common_threads.h"
8
9 #ifdef linux
10 #include <semaphore.h>
11 #elif __APPLE__
12 #include "zemaphore.h"
13 #endif
14
15 typedef struct _rwlock_t {
16     sem_t writelock;
17     sem_t lock;
18     int readers;
19 } rwlock_t;
20
21 void rwlock_init(rwlock_t *lock) {
22     lock->readers = 0;
23     Sem_init(&lock->lock, 1);
24     Sem_init(&lock->writelock, 1);

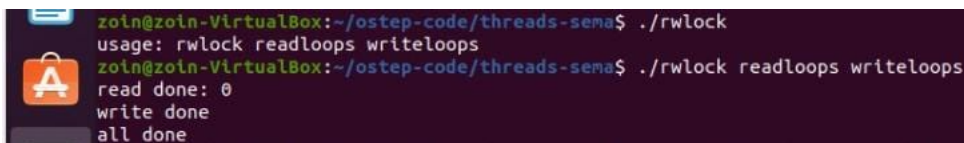
```

```
25 }
26
27 void rwlock_acquire_readlock(rwlock_t *lock) {
28     Sem_wait(&lock->lock);
29     lock->readers++;
30     if (lock->readers == 1)
31         Sem_wait(&lock->writelock);
32     Sem_post(&lock->lock);
33 }
34
35 void rwlock_release_readlock(rwlock_t *lock) {
36     Sem_wait(&lock->lock);
37     lock->readers--;
38     if (lock->readers == 0)
39         Sem_post(&lock->writelock);
40     Sem_post(&lock->lock);
41 }
42
43 void rwlock_acquire_writelock(rwlock_t *lock) {
44     Sem_wait(&lock->writelock);
45 }
46
47 void rwlock_release_writelock(rwlock_t *lock) {
48     Sem_post(&lock->writelock);
49 }
50
51 int read_loops;
52 int write_loops;
53 int counter = 0;
54
55 rwlock_t mutex;
56
57 void *reader(void *arg) {
58     int i;
59     int local = 0;
60     for (i = 0; i < read_loops; i++) {
61         rwlock_acquire_readlock(&mutex);
62         local = counter;
63         rwlock_release_readlock(&mutex);
64         printf("read %d\n", local);
65     }
66     printf("read done: %d\n", local);
67     return NULL;
68 }
69
70 void *writer(void *arg) {
71     int i;
72     for (i = 0; i < write_loops; i++) {
73         rwlock_acquire_writelock(&mutex);
74         counter++;
75         rwlock_release_writelock(&mutex);
76     }
77     printf("write done\n");
78     return NULL;
79 }
80
81 int main(int argc, char *argv[]) {
82     if (argc != 3) {
83         fprintf(stderr, "usage: rwlock readloops writeloops\n");
84         exit(1);
85     }
86     read_loops = atoi(argv[1]);
```

```
87 write_loops = atoi(argv[2]);
88
89 rwlock_init(&mutex);
90 pthread_t c1, c2;
91 Pthread_create(&c1, NULL, reader, NULL);
92 Pthread_create(&c2, NULL, writer, NULL);
93 Pthread_join(c1, NULL);
94 Pthread_join(c2, NULL);
95 printf("all done\n");
96 return 0;
97 }
98
```

5.2 Output

Berikut hasil percobaan yang saya buat pada terminal :



```
zoin@zoin-VirtualBox:~/ostep-code/threads-sema$ ./rwlock
usage: rwlock readloops writeloops
zoin@zoin-VirtualBox:~/ostep-code/threads-sema$ ./rwlock readloops writeloops
read done: 0
write done
all done
```

Gambar 4: Membuat shell script di nano

5.3 Penjelasan Reader/writer Locks

Secara umumnya, semaphore writelock untuk memastikan hanya satu writer saja yang mendapatkan lock dan memperbaruistruktur datanya dengan masuknya ke critical section. Kondisi ketika lock didapatkan, reader yang pertama akan mendapatkan lock tersebut dan mulai menambahkan variabel pembaca agar dapat melacak berapa pembaca yang ada saat ini pada struktur data. Perlu diperhatikan bahwa rwlock acquire readlock terjadi saat pembaca ke-1 mendapatkan lock dan writelock dengan memanggil Sem wait pada saat semaphore writelock yang akan dilepaskan lock-nya saat memanggil Sem post.

Write process dan read process saling berpasangan hal ini akan membuat perubahan terutama pada proses di overhead karena sekiranya write process tertahan atau menunggu maka read process juga tertahan yang membuat bukan mempercepat, tapi memperlambat.

6 Dining Philosophers

6.1 Deadlock

BDining Philosophers adalah salah satu problem yang kemudian ditemukan Dijkstra yaitu idenya yang berupa logika 5 filsuf di sebuah meja makan berbentuk lingkaran, kemudian membuat perumpamaan para filsuf tersebut akan mengambil garpu di sisi kanan dan kirinya. Hal yang menjadi permasalahan adalah ketika dua filsuf yang berdampingan namun mengambil garpu yang sama. Oleh karena itu diperlukan fungsi bantu yang disebut left dan right. Kondisi ketika salah satu philosopher merunjuk pada suatu arah, ia akan mengajak bagian yang lain untuk mengatasi persoalan pada bagian lain

6.1.1 Source code

```

1  class SynthiaDataset(Dataset):
2
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <pthread.h>
7
8  #include "common.h"
9  #include "common_threads.h"
10
11 #ifdef linux
12 #include <semaphore.h>
13 #elif APPLE
14 #include "zemaphore.h"
15 #endif
16
17 typedef struct {
18     int num_loops;
19     int thread_id;
20 } arg_t;
21
22 sem_t forks[5];
23 sem_t print_lock;
24
25 void space(int s) {
26     Sem_wait(&print_lock);
27     int i;
28     for (i = 0; i < s * 10; i++)
29         printf(" ");
30 }
31
32 void space_end() {
33     Sem_post(&print_lock);
34 }
35
36 int left(int p) {
37     return p;
38 }
39
40 int right(int p) {
41     return (p + 1) % 5;
42 }
43
44 void get_forks(int p) {
45     space(p); printf("%d: try %d\n", p, left(p)); space_end();
46     Sem_wait(&forks[left(p)]);
47     space(p); printf("%d: try %d\n", p, right(p)); space_end();

```

```

48     Sem_wait(&forks[right(p)]);
49 }
50
51 void put_forks(int p) {
52     Sem_post(&forks[left(p)]);
53     Sem_post(&forks[right(p)]);
54 }
55
56 void think() {
57     return;
58 }
59
60 void eat() {
61     return;
62 }
63
64 void *philosopher(void *arg) {
65     arg_t *args = (arg_t *) arg;
66
67     space(args->thread_id); printf("%d: start\n", args->thread_id); space_end();
68
69     int i;
70     for (i = 0; i < args->num_loops; i++) {
71         space(args->thread_id); printf("%d: think\n", args->thread_id); space_end();
72         think();
73         get_forks(args->thread_id);
74         space(args->thread_id); printf("%d: eat\n", args->thread_id); space_end();
75         eat();
76         put_forks(args->thread_id);
77         space(args->thread_id); printf("%d: done\n", args->thread_id); space_end();
78     }
79     return NULL;
80 }
81
82 int main(int argc, char *argv[]) {
83     if (argc != 2) {
84         fprintf(stderr, "usage: dining_philosophers <num_loops>\n");
85         exit(1);
86     }
87     printf("dining: started\n");
88
89     int i;
90     for (i = 0; i < 5; i++)
91         Sem_init(&forks[i], 1);
92     Sem_init(&print_lock, 1);
93
94     pthread_t p[5];
95     arg_t a[5];
96     for (i = 0; i < 5; i++) {
97         a[i].num_loops = atoi(argv[1]);
98         a[i].thread_id = i;
99         Pthread_create(&p[i], NULL, philosopher, &a[i]);
100     }
101
102     for (i = 0; i < 5; i++)
103         Pthread_join(p[i], NULL);
104
105     printf("dining: finished\n");
106     return 0;
107 }
108

```

Kode 1: Source code

6.1.2 Output Print Deadlock

```

zohm@zohm-VirtualBox: ~/jostep-code/threads-sema
$ ./dining_philosophers_deadlock 3
dining: started
dining: finished
dining: started
4: start
4: think
4: try 4
4: try 0
4: eat
4: done
4: think
4: try 4
4: try 0
4: eat
4: done
4: think
4: try 4
4: try 0
4: eat
4: done
3: start
3: think
3: try 3
3: try 4
3: eat
3: done
3: think
3: try 3
3: try 4
3: eat
3: done
3: think

zohm@zohm-VirtualBox: ~/jostep-code/threads-sema
$ ./dining_philosophers_deadlock_print 3
3: eat
3: done
2: start
2: think
2: try 2
2: try 3
2: eat
2: done
2: think
2: try 2
2: try 3
2: eat
2: done
2: think
2: try 2
2: try 3
2: eat
2: done
1: start
1: think
1: try 1
1: try 2
1: eat
1: done
1: think
1: try 1
1: try 2
1: eat
1: done
1: think
1: try 1
1: try 2
1: eat
1: done
1: think
1: try 1
1: try 2
1: eat
1: done
0: start
0: think
0: try 0
0: try 1
0: eat
0: done
0: think
0: try 0
0: try 1
0: eat
0: done
0: think
0: try 0
0: try 1
0: eat
0: done
0: think
0: try 0
0: try 1
0: eat
0: done
dining: finished

```

6.2 Output

Berikut hasil percobaan yang saya buat pada terminal :

```

zoin@zoin-VirtualBox:~/ostep-code/threads-sema$ ./dining_philosophers_deadlock
usage: dining_philosophers_deadlock <num_loops>
zoin@zoin-VirtualBox:~/ostep-code/threads-sema$ ./dining_philosophers_deadlock_print
usage: dining_philosophers <num_loops>
zoin@zoin-VirtualBox:~/ostep-code/threads-sema$ ./dining_philosophers_deadlock
usage: dining_philosophers_deadlock <num_loops>
zoin@zoin-VirtualBox:~/ostep-code/threads-sema$ ./dining_philosophers_no_deadlock
usage: dining_philosophers <num_loops>
zoin@zoin-VirtualBox:~/ostep-code/threads-sema$ ./dining_philosophers_no_deadlock_print
usage: dining_philosophers <num_loops>
zoin@zoin-VirtualBox:~/ostep-code/threads-sema$

```

6.2.1 Penjelasan Dining Philosophers Deadlock

Pada implementasi program Dining Philosopher Deadlock mempunyai cerita yang menarik dibelakangnya. Terdapat suatu masalah konkurensi yang dulu terkenal yang hanya diselesaikan oleh Dijkstra, masalah tersebut terkenal karena seru dan menarik secara intelektual yaitu dengan nama Philosopher problem. Kondisinya ketika 5 philoshoper yang duduk mengelilingi meja bundar terdapat sepasang philosopher single fork yang mana untuk memulai makan dibutuhkan sepasang forks satu di sebelah kanan dan satu di sebelah kiri. Dengan adanya solusi oleh Downey, diperlukannya beberapa fungsi bantu yang disebut left dan right.

Kondisi ketika philosopher P diminta agar merujuk ke fork kiri akan memulai memanggil fungsi left, dan sebaliknya jika diminta merujuk ke fork kanan akan memanggil fungsi right. Terdapat modulo yang mana menangani satu persoalan yaitu philosopher akhir dengan P sama dengan 4 mengambil fork bagian kanan saat fork bernilai kosong.

6.3 No Deadlock

6.3.1 Source code

```

1
2  class SynthiaDataset(Dataset):
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <pthread.h>
7
8  #include "common.h"
9  #include "common_threads.h"
10
11 #ifdef linux
12 #include <semaphore.h>
13 #elif APPLE
14 #include "zemaphore.h"
15 #endif
16
17 typedef struct {
18     int num_loops;
19     int thread_id;
20 } arg_t;
21
22 sem_t forks[5];
23 sem_t print_lock;
24
25 void space(int s) {
26     Sem_wait(&print_lock);
27     int i;

```

```

28     for (i = 0; i < s * 10; i++)
29         printf(" ");
30     }
31
32     void space_end() {
33         Sem_post(&print_lock);
34     }
35
36     int left(int p) {
37         return p;
38     }
39
40     int right(int p) {
41         return (p + 1) % 5;
42     }
43
44     void get_forks(int p) {
45         if (p == 4) {
46             space(p); printf("4 try %d\n", right(p)); space_end();
47             Sem_wait(&forks[right(p)]);
48             space(p); printf("4 try %d\n", left(p)); space_end();
49             Sem_wait(&forks[left(p)]);
50         } else {
51             space(p); printf("try %d\n", left(p)); space_end();
52             Sem_wait(&forks[left(p)]);
53             space(p); printf("try %d\n", right(p)); space_end();
54             Sem_wait(&forks[right(p)]);
55         }
56     }
57
58     void put_forks(int p) {
59         Sem_post(&forks[left(p)]);
60         Sem_post(&forks[right(p)]);
61     }
62
63     void think() {
64         return;
65     }
66
67     void eat() {
68         return;
69     }
70
71     void *philosopher(void *arg) {
72         arg_t *args = (arg_t *) arg;
73
74         space(args->thread_id); printf("%d: start\n", args->thread_id); space_end();
75
76         int i;
77         for (i = 0; i < args->num_loops; i++) {
78             space(args->thread_id); printf("%d: think\n", args->thread_id); space_end();
79             think();
80             get_forks(args->thread_id);
81             space(args->thread_id); printf("%d: eat\n", args->thread_id); space_end();
82             eat();
83             put_forks(args->thread_id);
84             space(args->thread_id); printf("%d: done\n", args->thread_id); space_end();
85         }
86         return NULL;
87     }
88
89     int main(int argc, char *argv[]) {

```

```
90     if (argc != 2) {
91         fprintf(stderr, "usage: dining_philosophers <num_loops>\n");
92         exit(1);
93     }
94     printf("dining: started\n");
95
96     int i;
97     for (i = 0; i < 5; i++)
98         Sem_init(&forks[i], 1);
99     Sem_init(&print_lock, 1);
100
101     pthread_t p[5];
102     arg_t a[5];
103     for (i = 0; i < 5; i++) {
104         a[i].num_loops = atoi(argv[1]);
105         a[i].thread_id = i;
106         Pthread_create(&p[i], NULL, philosopher, &a[i]);
107     }
108
109     for (i = 0; i < 5; i++)
110         Pthread_join(p[i], NULL);
111
112     printf("dining: finished\n");
113     return 0;
114 }
115
```

6.3.2 Output Print No Deadlock

```

zoin@zoin-VirtualBox:~/jostep-code/threads-sema$ ./dining_philosophers_no_deadlock 4
dining: started
dining: finished
zoin@zoin-VirtualBox:~/jostep-code/threads-sema$ ./dining_philosophers_no_deadlock_print 4
dining: started
0: start
0: think
try 0
try 1
0: eat
0: done
0: think
try 0
try 1
0: eat
0: done
0: think
try 0
try 1
0: eat
0: done
0: think
1: start
1: think
try 1
try 2
1: eat
1: done
1: think
2: start
2: think
try 2
try 3
2: eat
2: done
2: think
try 2
try 3
2: eat
2: done
2: think
try 2
try 3
2: eat
2: done
2: think
3: start
3: think
try 3
try 4
3: eat
3: done
3: think
try 3
try 4
3: eat
3: done
3: think
4: start
4: think
try 0
try 4
4: eat
4: done
4: think
try 0
try 4
4: eat
4: done
4: think
try 0
try 4
4: eat
4: done
4: think
dining: finished
zoin@zoin-VirtualBox:~/jostep-code/threads-sema$

```

6.3.3 Penjelasan Dining Philosophers No Deadlock

Pada implementasi program Dining Philosophers No Deadlock di atas menjelaskan tentang percobaan menginisialisasi setiap semaphore di fork array agar bernilai 1. Perlu kita ketahui bahwa philosopher mempunyai angka dan juga kita bisa menuliskan get forks dan put forks secara terus-menerus. Kemudian, kita memerlukan lock untuk mendapati forks dengan mendapatkan forks sebelah kiri dan dilanjutkan sebelah kanan. Kemudian pastinya ketika kita selesai memakainya pasti akan kita lepaskan, tetapi kondisi tersebut tidak terjadi karena adanya deadlock. Apabila setiap philosopher mengambil fork sebelah kiri terlebih dahulu sebelum bisa mengambil fork sebelah kanan, maka menyebabkan stuck dan dapat menahan satu fork yang mana membuat fork lainnya menunggu untuk selamanya. Contoh dari penggambarannya ialah misalkan philosopher 0 mengambil fork 0, philosopher 1 mengambil fork 1, philosopher 2 mengambil fork 2, dan philosopher 3

mengambil fork 3 akan menyebabkan semua philosopher terperangkap atau stuck karena tidak ada ruangan pada philosopher.

Dari permasalahan tersebut Dijkstra menemukan solusinya dengan mengganti fork mendapatkan satu philosopher saja. Dengan asumsi 4 philosopher mengambil fork melalui aturan yang berbeda dengan sebelumnya, hal tersebut disebabkan karena philosopher akhir mengambil sebelah kanan terlebih dahulu sebelum sebelah kiri. Pasalnya, tidak ada aturan philosopher mengambil satu fork dan mengalami stuck dengan menunggu fork lainnya. Oleh karena itu, adanya siklus waiting ini merusak prosesnya.

7 Kesimpulan

Synchronisation and Deadlock bertujuan agar kita dapat mengimplementasikan algoritma semaphores pada program yang akan dibuat. kita juga diharuskan untuk memahami Fork/Join, Binary Semaphores, Producer/-Consumer, Reader/Writer Locks dan Dining Philosophers

Pada hands On 2 ini, ketika kita mengerjakan materi Synchronisation and Deadlock membuat kita lebih paham dengan adanya pemberian kode program dari suatu programmer yang membuat programnya sesuai dengan implementasi materi tersebut.

8 Link GitHub

- Berikut Link GitHub saya terkait Tugas Hands on 2 : [Klik di sini](#)