



Nama: Irwanto Yezekiel Sihotang (120140227)

Tugas Ke: 2

Mata Kuliah: Sistem Operasi RA (IF2223)

Tanggal: 11 April 2022

1 Tujuan Hands On 2

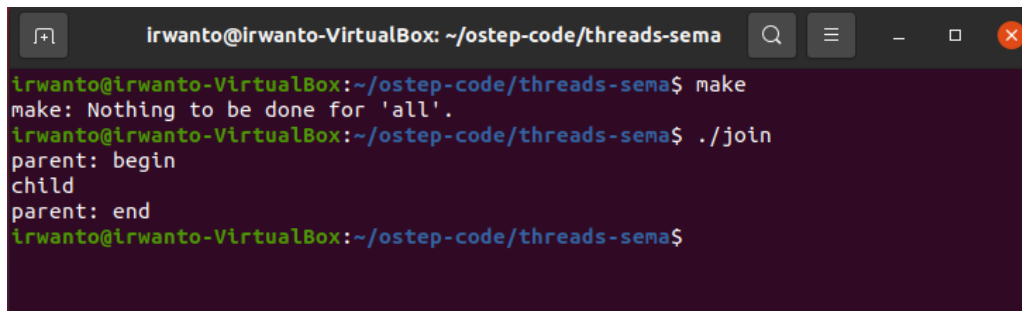
tujuan dari tugas Hands On dua ini adalah Agar memahami sistem sinkronisasi dan permasalahan yang ada serta Memahami solusi dalam menangani critical section. adapun hal yang harus dipahami dalam tugas Hands On dua ini adalah implementasi dari join menggunakan semaphores, Binary Semaphores, Producer Consumer, Reader / Writer Locks dan Dining Philosophers.

2 Fork/Join

2.1 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 sem_t s;
16
17 void *child(void *arg) {
18     sleep(2);
19     printf("child\n");
20     Sem_post(&s); // signal here: child is done
21     return NULL;
22 }
23
24 int main(int argc, char *argv[]) {
25     Sem_init(&s, 0);
26     printf("parent: begin\n");
27     pthread_t c;
28     Pthread_create(&c, NULL, child, NULL);
29     Sem_wait(&s); // wait here for child
30     printf("parent: end\n");
31     return 0;
32 }
33
```

2.2 Output



```
irwanto@irwanto-VirtualBox: ~/ostep-code/threads-sema$ make
make: Nothing to be done for 'all'.
irwanto@irwanto-VirtualBox:~/ostep-code/threads-sema$ ./join
parent: begin
child
parent: end
irwanto@irwanto-VirtualBox:~/ostep-code/threads-sema$
```

Gambar 1: Fork/Join

2.3 Penjelasan Fork/Join

semaphore adalah suatu struktur data komputer yang berguna untuk sinkronisasi proses dan berfungsi memerintahkan program agar menjalankan proses. contohnya adalah *thread* menunggu *list* sehingga *list* menjadi berisi atau tidak kosong. Dalam kondisi tersebut, semaphore akan didefinisikan dan dimulai ke 0 oleh Sem init. Tujuan dari proses ini adalah semaphore akan dibagikan di antara *thread* dalam proses yang sama. Kemudian ketika pembuatan *thread* selesai, akan terus memanggil fungsi child semaphore yang akan memberi sinyal bahwa proses child selesai dan dimulai kembali. Ketika child Setelah selesai, semaphore akan melanjutkan dan menampilkan "parent : end".

3 Binary Semaphores

3.1 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 sem_t mutex;
16 volatile int counter = 0;
17
18 void *child(void *arg) {
19     int i;
20     for (i = 0; i < 10000000; i++) {
21         Sem_wait(&mutex);
22         counter++;
23         Sem_post(&mutex);
24     }
25     return NULL;
26 }
27
```

```

28 int main(int argc, char *argv[]) {
29     Sem_init(&mutex, 1);
30     pthread_t c1, c2;
31     Pthread_create(&c1, NULL, child, NULL);
32     Pthread_create(&c2, NULL, child, NULL);
33     Pthread_join(c1, NULL);
34     Pthread_join(c2, NULL);
35     printf("result: %d (should be 20000000)\n", counter);
36     return 0;
37 }
38

```

Kode 1: Code Binary semaphores

3.2 Output



```

irwanto@irwanto-VirtualBox:~/ostep-code/threads-sema$ ./binary
result: 20000000 (should be 20000000)

```

Gambar 2: Binary Semaphores

3.3 Penjelasan Binary Semaphores

Pada kode di atas terdapat variabel *sem t mutex* atau bisa disebut *mutual exclusion* berperan dalam mengelola penggunaan *resource*. Mutex ada untuk mencegah kondisi balapan. Pertama kita set dan inisialisasi semaphore mutex dengan nilai 1, Selanjutnya, sebuah *thread* dibuat dengan inisial c1 dan c2, yang berguna dalam menjalankan fungsi *child*. Kemudian i akan menginisialisasi dalam *loop* sampai nilai i kurang dari 10000000 untuk dieksekusi *sem wait* dan pada saat itu *value* akan berkurang dan *critical section* akan dimulai, akan ada tambahan nilai *counter* yang kemudian semaphore akan memproses panggilan untuk menaikkan nilai semaphore sebagai sinyal bahwa *critical section* sudah selesai. Kemudian program akan mengulangi prosesnya sampai kondisi terpenuhi, *thread* c2 akan terus menjalankan fungsi *child*. Ketika selesai, akan memulai *return* fungsi *main* yang akan menampilkan hasil penghitung dieksekusi menampilkan output dalam nilai 20000000.

4 Producer Consumer

4.1 Source Code

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <assert.h>
4 #include <pthread.h>
5 #include <stdlib.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 int max;
17 int loops;

```

```
18 int *buffer;
19
20 int use = 0;
21 int fill = 0;
22
23 sem_t empty;
24 sem_t full;
25 sem_t mutex;
26
27 #define CMAX (10)
28 int consumers = 1;
29
30 void do_fill(int value) {
31     buffer[fill] = value;
32     fill++;
33     if (fill == max)
34         fill = 0;
35 }
36
37 int do_get() {
38     int tmp = buffer[use];
39     use++;
40     if (use == max)
41         use = 0;
42     return tmp;
43 }
44
45 void *producer(void *arg) {
46     int i;
47     for (i = 0; i < loops; i++) {
48         Sem_wait(&empty);
49         Sem_wait(&mutex);
50         do_fill(i);
51         Sem_post(&mutex);
52         Sem_post(&full);
53     }
54
55     // end case
56     for (i = 0; i < consumers; i++) {
57         Sem_wait(&empty);
58         Sem_wait(&mutex);
59         do_fill(-1);
60         Sem_post(&mutex);
61         Sem_post(&full);
62     }
63
64     return NULL;
65 }
66
67 void *consumer(void *arg) {
68     int tmp = 0;
69     while (tmp != -1) {
70         Sem_wait(&full);
71         Sem_wait(&mutex);
72         tmp = do_get();
73         Sem_post(&mutex);
74         Sem_post(&empty);
75         printf("%lld %d\n", (long long int) arg, tmp);
76     }
77     return NULL;
78 }
79
```

```

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

```

Kode 2: source code Producer Consumer

4.2 Output

4.3 Penjelasan Producer Consumer

Implementasi *producer/konsumer* disebut *bounded buffer*. Isi dari program ini adalah untuk memanggil, mengurangi, menghalangi consumer dan berharap *thread* lainnya dapat memanggil *Sem post* ketika sudah penuh. Kemudian program akan memulai fungsi prosedur yang berguna untuk memanggil *sem wait(empty)* dan *sem post (mutex)*. *Producer* akan diisi dengan fungsi *do fill* pada input pertama *buffer* setelah *empty* dikurangi menjadi nilai 0. Kemudian *Producer* akan terus berjalan sampai panggilan *sem post(mutex)* dan *sem post(full)* yang akan merubah nilai total dari -1 menjadi 0. Jadi Konsumen akan mengulang dan memblok dengan *value semaphore* kosong.

5 Reader / Writer Locks

5.1 Source Code

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <assert.h>
4 #include <pthread.h>
5 #include <stdlib.h>
6
7 #include "common.h"

```

```

irwanto@irwanto-VirtualBox:~/ostep-code/threads-sema$ ./binary
result: 20000000 (should be 20000000)
irwanto@irwanto-VirtualBox:~/ostep-code/threads-sema$ ./producer_consumer_work 1 1000 1
bash: ./producer_consumer_work: No such file or directory
irwanto@irwanto-VirtualBox:~/ostep-code/threads-sema$ ./producer_consumer_works 1 1000 1
0 0
0 1
0 2
0 3
0 4
0 5
0 6
0 7
0 8
0 9
0 10
0 11
0 12
0 13
0 14
0 15
0 16
0 17
0 18
0 19
0 20
0 21
0 22
0 23
0 24
0 25
0 26
0 27
0 28
0 29
0 30
0 31
0 32
0 33
0 34
0 35
0 36
0 37
0 38
0 39
0 40
0 41
0 42
0 43
0 44

```

Gambar 3: Producer Consumer

```

8 #include "common_threads.h"
9
10 #ifdef linux
11 #include <semaphore.h>
12 #elif __APPLE__
13 #include "zemaphore.h"
14 #endif
15
16 int max;
17 int loops;
18 int *buffer;
19
20 int use = 0;
21 int fill = 0;
22
23 sem_t empty;
24 sem_t full;
25 sem_t mutex;
26
27 #define CMAX (10)

```

```
28 int consumers = 1;
29
30 void do_fill(int value) {
31     buffer[fill] = value;
32     fill++;
33     if (fill == max)
34         fill = 0;
35 }
36
37 int do_get() {
38     int tmp = buffer[use];
39     use++;
40     if (use == max)
41         use = 0;
42     return tmp;
43 }
44
45 void *producer(void *arg) {
46     int i;
47     for (i = 0; i < loops; i++) {
48         Sem_wait(&empty);
49         Sem_wait(&mutex);
50         do_fill(i);
51         Sem_post(&mutex);
52         Sem_post(&full);
53     }
54
55     // end case
56     for (i = 0; i < consumers; i++) {
57         Sem_wait(&empty);
58         Sem_wait(&mutex);
59         do_fill(-1);
60         Sem_post(&mutex);
61         Sem_post(&full);
62     }
63
64     return NULL;
65 }
66
67 void *consumer(void *arg) {
68     int tmp = 0;
69     while (tmp != -1) {
70         Sem_wait(&full);
71         Sem_wait(&mutex);
72         tmp = do_get();
73         Sem_post(&mutex);
74         Sem_post(&empty);
75         printf("%lld %d\n", (long long int) arg, tmp);
76     }
77     return NULL;
78 }
79
80 int main(int argc, char *argv[]) {
81     if (argc != 4) {
82         fprintf(stderr, "usage: %s <buffersize> <loops> <consumers>\n", argv[0]);
83         exit(1);
84     }
85     max = atoi(argv[1]);
86     loops = atoi(argv[2]);
87     consumers = atoi(argv[3]);
88     assert(consumers <= CMAX);
89 }
```

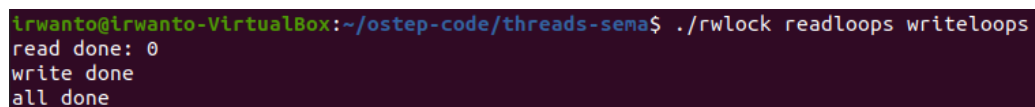
```

90  buffer = (int *) malloc(max * sizeof(int));
91  assert(buffer != NULL);
92  int i;
93  for (i = 0; i < max; i++) {
94  buffer[i] = 0;
95  }
96
97  Sem_init(&empty, max); // max are empty
98  Sem_init(&full, 0);    // 0 are full
99  Sem_init(&mutex, 1);   // mutex
100
101  pthread_t pid, cid[CMAX];
102  Pthread_create(&pid, NULL, producer, NULL);
103  for (i = 0; i < consumers; i++) {
104  Pthread_create(&cid[i], NULL, consumer, (void *) (long long int) i);
105  }
106  Pthread_join(pid, NULL);
107  for (i = 0; i < consumers; i++) {
108  Pthread_join(cid[i], NULL);
109  }
110  return 0;
111 }
112

```

Kode 3: source code Reader / writer locks

5.2 Output



```

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

```

Gambar 4: Reader / Writer Locks

5.3 Penjelasan Reader / Writer Locks

Secara umum, *semaphore writelock* untuk memastikan bahwa hanya satu *writer* yang di *lock* dan perbarui struktur data untuk menyertakan *critical section*. Keadaan di mana *lock* diperoleh, *reader* yang pertama men-*lock* dan mulai menambahkan variabel pembaca untuk melacak berapa banyak pembaca saat ini di struktur data. Perhatikan bahwa *rwlock acquire readlock* terjadi ketika pembaca pertama mendapatkan *lock* dan menulis *lock* dengan Tidak menunggu semaphore *writelock* dilepaskan saat memanggil *Sem post*.

6 Dining Philosophers

6.1 Deadlock

6.1.1 Source Code

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4
5 #include "common.h"
6 #include "common_threads.h"
7

```



```
8 #ifdef linux
9 #include <semaphore.h>
10 #elif __APPLE__
11 #include "zemaphore.h"
12 #endif
13
14 typedef struct {
15     int num_loops;
16     int thread_id;
17 } arg_t;
18
19 sem_t forks[5];
20 sem_t print_lock;
21
22 void space(int s) {
23     Sem_wait(&print_lock);
24     int i;
25     for (i = 0; i < s * 10; i++)
26         printf(" ");
27 }
28
29 void space_end() {
30     Sem_post(&print_lock);
31 }
32
33 int left(int p) {
34     return p;
35 }
36
37 int right(int p) {
38     return (p + 1) % 5;
39 }
40
41 void get_forks(int p) {
42     space(p); printf("%d: try %d\n", p, left(p)); space_end();
43     Sem_wait(&forks[left(p)]);
44     space(p); printf("%d: try %d\n", p, right(p)); space_end();
45     Sem_wait(&forks[right(p)]);
46 }
47
48 void put_forks(int p) {
49     Sem_post(&forks[left(p)]);
50     Sem_post(&forks[right(p)]);
51 }
52
53 void think() {
54     return;
55 }
56
57 void eat() {
58     return;
59 }
60
61 void *philosopher(void *arg) {
62     arg_t *args = (arg_t *) arg;
63
64     space(args->thread_id); printf("%d: start\n", args->thread_id); space_end();
65
66     int i;
67     for (i = 0; i < args->num_loops; i++) {
68         space(args->thread_id); printf("%d: think\n", args->thread_id); space_end();
69         think();
```

```
70 get_forks(args->thread_id);
71 space(args->thread_id); printf("%d: eat\n", args->thread_id); space_end();
72 eat();
73 put_forks(args->thread_id);
74 space(args->thread_id); printf("%d: done\n", args->thread_id); space_end();
75 }
76 return NULL;
77 }
78
79 int main(int argc, char *argv[]) {
80     if (argc != 2) {
81         fprintf(stderr, "usage: dining_philosophers <num_loops>\n");
82         exit(1);
83     }
84     printf("dining: started\n");
85
86     int i;
87     for (i = 0; i < 5; i++)
88         Sem_init(&forks[i], 1);
89     Sem_init(&print_lock, 1);
90
91     pthread_t p[5];
92     arg_t a[5];
93     for (i = 0; i < 5; i++) {
94         a[i].num_loops = atoi(argv[1]);
95         a[i].thread_id = i;
96         Pthread_create(&p[i], NULL, philosopher, &a[i]);
97     }
98
99     for (i = 0; i < 5; i++)
100         Pthread_join(p[i], NULL);
101
102     printf("dining: finished\n");
103     return 0;
104 }
105
```

Kode 4: source code deadlock

6.1.2 Output

```

irwanto@irwanto-VirtualBox:~/ostep-code/threads-sema$ ./dining_philosophers_deadlock 3
dining: started
dining: finished
irwanto@irwanto-VirtualBox:~/ostep-code/threads-sema$ ./dining_philosophers_deadlock_print 3
dining: started
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
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
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

```

Gambar 5: Dining Philosophers deadlock

6.1.3 Penjelasan Dining Philosophers deadlock

Dalam pelaksanaan program *Dining Philosopher Deadlock*, ada cerita menarik di baliknya. Ada masalah konkurensi yang dulu terkenal yang hanya bisa diselesaikan oleh Dijkstra, masalah yang terkenal lucu dan menarik secara intelektual, yaitu masalah *Philosopher's*. kondisi Ketika 5 *philosopher* duduk mengelilingi meja bundar, ada sepasang *philosopher single fork* yang dimana untuk memulai makan membutuhkan sepasang *fork* satu di kanan dan satu di kiri. Dengan solusi Downey, dibutuhkan beberapa fungsi pembantu yang disebut *left* dan *right*. Kondisi di mana *philosopher* P diminta untuk merujuk ke *fork* kiri akan mulai memanggil fungsi *left*, dan jika tidak, jika diminta untuk merujuk ke *fork* yang benar, itu akan memanggil fungsi yang benar. Terdapat modulo yang mana menangani satu persoalan yaitu *philosopher* akhir dengan P sama dengan 4 mengambil fork bagian kanan saat fork bernilai kosong.

6.2 no Deadlock

6.2.1 Source Code

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4

```

```
5 #include "common.h"
6 #include "common_threads.h"
7
8 #ifdef linux
9 #include <semaphore.h>
10 #elif __APPLE__
11 #include "zemaphore.h"
12 #endif
13
14 typedef struct {
15     int num_loops;
16     int thread_id;
17 } arg_t;
18
19 sem_t forks[5];
20 sem_t print_lock;
21
22 void space(int s) {
23     Sem_wait(&print_lock);
24     int i;
25     for (i = 0; i < s * 10; i++)
26         printf(" ");
27 }
28
29 void space_end() {
30     Sem_post(&print_lock);
31 }
32
33 int left(int p) {
34     return p;
35 }
36
37 int right(int p) {
38     return (p + 1) % 5;
39 }
40
41 void get_forks(int p) {
42     if (p == 4) {
43         space(p); printf("4 try %d\n", right(p)); space_end();
44         Sem_wait(&forks[right(p)]);
45         space(p); printf("4 try %d\n", left(p)); space_end();
46         Sem_wait(&forks[left(p)]);
47     } else {
48         space(p); printf("try %d\n", left(p)); space_end();
49         Sem_wait(&forks[left(p)]);
50         space(p); printf("try %d\n", right(p)); space_end();
51         Sem_wait(&forks[right(p)]);
52     }
53 }
54
55 void put_forks(int p) {
56     Sem_post(&forks[left(p)]);
57     Sem_post(&forks[right(p)]);
58 }
59
60 void think() {
61     return;
62 }
63
64 void eat() {
65     return;
66 }
```

```
67
68 void *philosopher(void *arg) {
69     arg_t *args = (arg_t *) arg;
70
71     space(args->thread_id); printf("%d: start\n", args->thread_id); space_end();
72
73     int i;
74     for (i = 0; i < args->num_loops; i++) {
75         space(args->thread_id); printf("%d: think\n", args->thread_id); space_end();
76         think();
77         get_forks(args->thread_id);
78         space(args->thread_id); printf("%d: eat\n", args->thread_id); space_end();
79         eat();
80         put_forks(args->thread_id);
81         space(args->thread_id); printf("%d: done\n", args->thread_id); space_end();
82     }
83     return NULL;
84 }
85
86 int main(int argc, char *argv[]) {
87     if (argc != 2) {
88         fprintf(stderr, "usage: dining_philosophers <num_loops>\n");
89         exit(1);
90     }
91     printf("dining: started\n");
92
93     int i;
94     for (i = 0; i < 5; i++)
95         Sem_init(&forks[i], 1);
96     Sem_init(&print_lock, 1);
97
98     pthread_t p[5];
99     arg_t a[5];
100     for (i = 0; i < 5; i++) {
101         a[i].num_loops = atoi(argv[1]);
102         a[i].thread_id = i;
103         Pthread_create(&p[i], NULL, philosopher, &a[i]);
104     }
105
106     for (i = 0; i < 5; i++)
107         Pthread_join(p[i], NULL);
108
109     printf("dining: finished\n");
110     return 0;
111 }
112
```

Kode 5: source code no deadlock

8 link GitHub

berikut saya lampirkan link GitHub :

- (Sumber: [tugas Handson 2](#)).