

Dhvanil Shah  
ECE 357  
Prof. Hakner  
14 December 2018

## Problem Set 6

### Table of Contents

<b>Write Up</b>	<b>1</b>
<b>Code</b>	
<i>Makefile</i>	4
<i>tas64.S (MacOS)</i>	5
<i>spin.c</i>	6
<i>spin.h</i>	7
<i>sem.c</i>	8
<i>sem.h</i>	10
<i>fifo.c</i>	11
<i>fifo.h</i>	13
<i>testTAS.c</i>	14
<i>testFIFO.c</i>	16

## Write Up

Testing with **one reader and one writer** using 20 iterations. Each write and each read is sequentially outputted to two separate files. The “diff” command is used to indicate the write and read order is the same.

```
[Dhvanils-MacBook-Pro:prog6 dhvanil$ ./testFIFO
WRITE 711410000 by PID: 71141
WRITE 711410001 by PID: 71141
WRITE 711410002 by PID: 71141
WRITE 711410003 by PID: 71141
WRITE 711410004 by PID: 71141
WRITE 711410005 by PID: 71141
WRITE 711410006 by PID: 71141
WRITE 711410007 by PID: 71141
WRITE 711410008 by PID: 71141
WRITE 711410009 by PID: 71141
WRITE 711410010 by PID: 71141
WRITE 711410011 by PID: 71141
WRITE 711410012 by PID: 71141
WRITE 711410013 by PID: 71141
WRITE 711410014 by PID: 71141
WRITE 711410015 by PID: 71141
WRITE 711410016 by PID: 71141
WRITE 711410017 by PID: 71141
WRITE 711410018 by PID: 71141
WRITE 711410019 by PID: 71141
READ 711410000 by PID: 71142
READ 711410001 by PID: 71142
READ 711410002 by PID: 71142
READ 711410003 by PID: 71142
READ 711410004 by PID: 71142
READ 711410005 by PID: 71142
READ 711410006 by PID: 71142
READ 711410007 by PID: 71142
READ 711410008 by PID: 71142
READ 711410009 by PID: 71142
READ 711410010 by PID: 71142
READ 711410011 by PID: 71142
READ 711410012 by PID: 71142
READ 711410013 by PID: 71142
READ 711410014 by PID: 71142
READ 711410015 by PID: 71142
READ 711410016 by PID: 71142
READ 711410017 by PID: 71142
READ 711410018 by PID: 71142
READ 711410019 by PID: 71142
[Dhvanils-MacBook-Pro:prog6 dhvanil$ diff writes.txt reads.txt
Dhvanils-MacBook-Pro:prog6 dhvanil$ █
```

Testing with **multiple writes and one reader using many iterations**. Output is truncated but diff passes.

```
prog6 — -bash — 115x46
~/documents/os/prog6 — -bash
READ 710980016 by PID: 71101
READ 710980017 by PID: 71101
READ 710980018 by PID: 71101
READ 710980019 by PID: 71101
READ 710990000 by PID: 71101
READ 710990001 by PID: 71101
READ 710990002 by PID: 71101
READ 710990003 by PID: 71101
READ 710990004 by PID: 71101
READ 710990005 by PID: 71101
READ 710990006 by PID: 71101
READ 710990007 by PID: 71101
READ 710990008 by PID: 71101
READ 710990009 by PID: 71101
READ 710990010 by PID: 71101
READ 710990011 by PID: 71101
READ 710990012 by PID: 71101
READ 710990013 by PID: 71101
READ 710990014 by PID: 71101
READ 710990015 by PID: 71101
READ 710990016 by PID: 71101
READ 710990017 by PID: 71101
READ 710990018 by PID: 71101
READ 710990019 by PID: 71101
READ 711000000 by PID: 71101
READ 711000001 by PID: 71101
READ 711000002 by PID: 71101
READ 711000003 by PID: 71101
READ 711000004 by PID: 71101
READ 711000005 by PID: 71101
READ 711000006 by PID: 71101
READ 711000007 by PID: 71101
READ 711000008 by PID: 71101
READ 711000009 by PID: 71101
READ 711000010 by PID: 71101
READ 711000011 by PID: 71101
READ 711000012 by PID: 71101
READ 711000013 by PID: 71101
READ 711000014 by PID: 71101
READ 711000015 by PID: 71101
READ 711000016 by PID: 71101
READ 711000017 by PID: 71101
READ 711000018 by PID: 71101
READ 711000019 by PID: 71101
Dhvanils-MacBook-Pro:prog6 dhvanil$ diff writes.txt reads.txt
Dhvanils-MacBook-Pro:prog6 dhvanil$
```

Testing by **changing the order of lock and unlock** in `sem_inc()`. The program hangs, indicating that it is broken.

```
[Dhvanils-MacBook-Pro:prog6 dhvanil$ make testFIFO
Building 'testFIFO.o'...
Building 'tas.o'...
Building 'sem.o'...
Building 'fifo.o'...
Building 'spin.o'...
Building 'testTAS'...
[Dhvanils-MacBook-Pro:prog6 dhvanil$ ./testFIFO
WRITE 717690000 by PID: 71769
```

```
1 .PHONY: clean
2
3 testFIFO: testFIFO.o tas64.o sem.o fifo.o spin.o
4     @echo "Building 'testTAS'..."
5     @gcc -o testFIFO testFIFO.o tas64.o sem.o fifo.o spin.o
6
7 testTAS: testTAS.o tas64.o spin.o
8     @echo "Building 'testTAS'..."
9     @gcc -o testTAS testTAS.o tas64.o spin.o
10
11 testTAS.o: testTAS.c
12     @echo "Building 'testTAS.o'..."
13     @gcc -c testTAS.c
14
15 spin.o: spin.c spin.h
16     @echo "Building 'spin.o'..."
17     @gcc -c spin.c
18
19 sem.o: sem.c sem.h
20     @echo "Building 'sem.o'..."
21     @gcc -c sem.c
22
23 fifo.o: fifo.c fifo.h
24     @echo "Building 'fifo.o'..."
25     @gcc -c fifo.c
26
27 testFIFO.o: testFIFO.c
28     @echo "Building 'testFIFO.o'..."
29     @gcc -c testFIFO.c
30
31 tas64.o: tas64.S
32     @echo "Building 'tas.o'..."
33     @gcc -c tas64.S
34
35
36 clean:
37     @echo "Removing all built files..."
38     @rm -f *.o testTAS testFIFO
39     @clear
```

```
1  .text
2  .globl _tas
3  _tas:
4      pushq %rbp
5      movq %rsp, %rbp
6      movq $1, %rax
7  #APP
8      lock;xchgb %al, (%rdi)
9  #NO_APP
10     movsbq %al, %rax
11     pop %rbp
12     ret
13 Lfe1:
```

```
1 #include "spin.h"
2 #include "tas.h"
3 #include <sched.h>
4
5 void spin_lock(volatile char *lock)
6 {
7     while (tas(lock))
8         sched_yield(); // No need to check sched yeild because it always
                          // succeeds in the linux implementation
9 }
10
11 void spin_unlock(volatile char *lock)
12 {
13     *lock = 0;
14 }
```

```
1 #ifndef __SPIN_H
2 void spin_lock(volatile char *lock);
3 void spin_unlock(volatile char *lock);
4 #define __SPIN_H
5 #endif
```

```
1 #include "sem.h"
2 #include "spin.h"
3
4 static void dummy() //Dummy handler. Does nothing
5 {
6     // return;
7 }
8
9 void sem_init(struct sem *s, int count)
10 {
11     s->spinlock = 0;
12     s->max = count;
13     s->free = count;
14     s->procInd = -1;
15
16     sigfillset(&s->mask_block);
17     sigdelset(&s->mask_block, SIGUSR1);
18     signal(SIGUSR1, dummy); // Prevent the signal from killing the process
19 }
20
21 int sem_try(struct sem *s)
22 {
23     spin_lock(&s->spinlock);
24     if (s->free > 0)
25     {
26         s->free -= 1;
27         spin_unlock(&s->spinlock);
28         return 1;
29     }
30     else
31     {
32         spin_unlock(&s->spinlock);
33         return 0;
34     }
35 }
36
37 void sem_wait(struct sem *s)
38 {
39     for (;;)
40     {
41         spin_lock(&s->spinlock);
42
43         if (s->free > 0)
44         {
45             s->free -= 1;
46             spin_unlock(&s->spinlock);
47             break;
48         }
49         else
50         {
```



```
50     ^
51     s->proc_block[s->procInd] = procNum; // Put process on waitlist
52     s->procInd += 1; //Book keeping
53     spin_unlock(&s->spinlock);
54     sigsuspend(&s->mask_block); //Put process to sleep
55 }
56 }
57 }
58
59 void sem_inc(struct sem *s)
60 {
61     spin_lock(&s->spinlock);
62
63     s->free += 1; // Increment semaphore
64     if (s->free == 1)
65     {
66         while (s->procInd != -1) //Loop to wake up all processes when sem
67         becomes 1
68         {
69             kill(pid_table[s->proc_block[s->procInd]], SIGUSR1);
70             s->procInd -= 1; // Book keeping
71         }
72     }
73     spin_unlock(&s->spinlock);
74 }
```

```
1 #ifndef __SEM_H
2
3 #include <signal.h>
4
5 #define NUM_PROC 64
6 #define MYPROCS 4
7
8 extern int procNum;          // Current Process Index
9 extern pid_t *pid_table;    // Table of Process PIDS
10
11 struct sem
12 {
13     char spinlock;          // The Lock
14     int max;                // Max Resources Available
15     int free;               // Actual Available Resources
16     int procInd;            // Index of proc_block for book keeping
17     int proc_block[NUM_PROC]; //List of Blocking Processes
18     sigset_t mask_block;    //Mask for all signals but SIGUSR1
19 };
20
21 // Initialize the semaphore *s with the initial count. Initialize
22 // any underlying data structures. sem_init should only be called
23 // once in the program (per semaphore). If called after the
24 // semaphore has been used, results are unpredictable.
25 void sem_init(struct sem *s, int count);
26
27 // Attempt to perform the "P" operation (atomically decrement
28 // the semaphore). If this operation would block, return 0,
29 // otherwise return 1.
30 int sem_try(struct sem *s);
31
32 // Perform the P operation, blocking until successful.
33 void sem_wait(struct sem *s);
34
35 // Perform the V operation. If any other tasks were sleeping
36 // on this semaphore, wake them by sending a SIGUSR1 to their
37 // process id (which is not the same as the virtual processor number).
38 // If there are multiple sleepers (this would happen if multiple
39 // virtual processors attempt the P operation while the count is <1)
40 // then \fBall\fP must be sent the wakeup signal.
41 void sem_inc(struct sem *s);
42 #define __SEM_H
43 #endif
```

```
1 #include "fifo.h"
2
3 void fifo_init(struct fifo *f)
4 {
5     f->next_read = 0;
6     f->next_write = 0;
7     sem_init(&f->empty, MYFIFO_BUFSIZ);
8     sem_init(&f->full, 0);
9     sem_init(&f->mutex, 1);
10 }
11
12 void fifo_wr(struct fifo *f, unsigned long d)
13 {
14     for (;;)
15     {
16         sem_wait(&f->empty);
17
18         if (sem_try(&f->mutex))
19         {
20             f->buffer[f->next_write] = d;
21             f->next_write = (f->next_write + 1) % MYFIFO_BUFSIZ;
22             sem_inc(&f->mutex);
23             sem_inc(&f->full);
24             break;
25         }
26         else
27         {
28             sem_inc(&f->empty);
29         }
30     }
31 }
32
33 unsigned long fifo_rd(struct fifo *f)
34 {
35     unsigned long d;
36     for (;;)
37     {
38         sem_wait(&f->full);
39         if (sem_try(&f->mutex))
40         {
41             d = f->buffer[f->next_read];
42             f->next_read = (f->next_read + 1) % MYFIFO_BUFSIZ;
43             sem_inc(&f->mutex);
44             sem_inc(&f->empty);
45             break;
46         }
47         else
48         {
49             sem_inc(&f->full);
50         }
51     }
52 }
```

```
50     }  
51 }  
52 return d;  
53 }
```

```
1 #ifndef __FIFO_H
2 #include "sem.h"
3
4 #define MYFIFO_BUFSIZ 4096
5
6 struct fifo
7 {
8     unsigned long buffer[MYFIFO_BUFSIZ];
9     int next_read, next_write;
10    struct sem empty; // empty spots
11    struct sem full; // filled spots
12    struct sem mutex; // mutex for operations
13 };
14
15 // Initialize the shared memory FIFO *f including any
16 // required underlying initializations (such as calling sem_init)
17 // The FIFO will have a fifo length of MYFIFO_BUFSIZ elements,
18 // which should be a static #define in fifo.h (a value of 4K is
19 // reasonable).
20 void fifo_init(struct fifo *f);
21
22 // Enqueue the data word d into the FIFO, blocking
23 // unless and until the FIFO has room to accept it.
24 // Use the semaphore primitives to accomplish blocking and waking.
25 // Writing to the FIFO shall cause any and all processes that
26 // had been blocked because it was empty to wake up.
27 void fifo_wr(struct fifo *f, unsigned long d);
28
29 // Dequeue the next data word from the FIFO and return it.
30 // Block unless and until there are available words
31 // queued in the FIFO. Reading from the FIFO shall cause
32 // any and all processes that had been blocked because it was
33 // full to wake up.
34 unsigned long fifo_rd(struct fifo *f);
35
36 #define __FIFO_H
37 #endif
```

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <sys/mman.h>
5 #include <sys/wait.h>
6 #include <errno.h>
7 #include <string.h>
8 #include <ctype.h>
9
10 #include "spin.h"
11
12 #define NUM_PROC 64
13 #define MYPROCS 4
14 #define NUMITR 1000000
15
16 void throwError(char *message, char *file)
17 {
18     if (file)
19         fprintf(stderr, "%s [%s]: Error code %i: %s\n", message, file, errno,
20 strerror(errno));
21     else
22         fprintf(stderr, "%s\n", message);
23     exit(-1);
24 }
25
26 int main(int argc, char const *argv[])
27 {
28     int pid[MYPROCS], myPID = 0;
29
30     unsigned long long idealCt = (MYPROCS * NUMITR);
31     unsigned long long *counter = (unsigned long long *)mmap(NULL,
32 sizeof(unsigned long long), PROT_READ | PROT_WRITE, MAP_SHARED |
33 MAP_ANONYMOUS, -1, 0);
34     unsigned long long *counterTAS = (unsigned long long *)mmap(NULL,
35 sizeof(unsigned long long), PROT_READ | PROT_WRITE, MAP_SHARED |
36 MAP_ANONYMOUS, -1, 0);
37
38     char *lock = (char *)mmap(NULL, sizeof(char), PROT_READ | PROT_WRITE,
39 MAP_SHARED | MAP_ANONYMOUS, -1, 0);
40
41     for (int i = 0; i < MYPROCS; i++)
42     {
43         if ((pid[i] = fork()) < 0)
44         {
45             throwError("Error: Failed to fork process.", NULL);
46         }
47         else if (pid[i] == 0)
48         {
49             myPID = 0;
50         }
51     }
52 }
```

```
45         for (i = 0; i < NUMITR; i++)
46         {
47             *counter += 1;
48         }
49
50         for (i = 0; i < NUMITR; i++)
51         {
52             spin_lock(lock);
53             *counterTAS += 1;
54             spin_unlock(lock);
55         }
56         break;
57     }
58     else
59         myPID = 1;
60 }
61
62 if (myPID)
63 {
64     for (int i = 0; i < MYPROCS; i++)
65     {
66         if (waitpid(pid[i], NULL, 0) < 0)
67         {
68             throwError("Error: Unable to wait for child process to
complete", NULL);
69         }
70     }
71     // PRINT OUT RESULTS
72     fprintf(stderr, "IDEAL COUNT: %llu | NON-TAS COUNT: %llu | TAS COUNT:
%llu\n", idealCt, *counter, *counterTAS);
73
74     if ((munmap(counter, sizeof(unsigned long long)) < 0) ||
(munmap(counterTAS, sizeof(unsigned long long)) < 0))
75         throwError("Error: Unable to munmap counter(s)", 0);
76
77     if ((munmap(lock, sizeof(char)) < 0))
78         throwError("Error: Unable to munmap lock", 0);
79 }
80
81 return 0;
82 }
83
```

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <sys/mman.h>
5 #include <sys/wait.h>
6 #include <errno.h>
7 #include <string.h>
8 #include "fifo.h"
9
10 #define WRITERS 5
11 #define NUMITR 200
12
13 int procNum;
14 pid_t *pid_table;
15
16 void throwError(char *message, char *file)
17 {
18     if (file)
19         fprintf(stderr, "%s [%s]: Error code %i: %s\n", message, file, errno,
20 strerror(errno));
21     else
22         fprintf(stderr, "%s\n", message);
23     exit(-1);
24 }
25
26 int main(int argc, char **argv)
27 {
28     struct fifo *f;
29     int i, j;
30     unsigned long datum;
31     FILE *writes = fopen("writes.txt", "w"), *reads = fopen("reads.txt", "w");
32     if (writes == NULL || reads == NULL)
33         throwError("Error: Unable to open reads and writes log", NULL);
34
35     f = (struct fifo *)mmap(NULL, sizeof(struct fifo), PROT_READ | PROT_WRITE,
36 MAP_SHARED | MAP_ANONYMOUS, -1, 0);
37     pid_table = (pid_t *)mmap(NULL, ((sizeof(pid_t)) * NUM_PROC), PROT_READ |
38 PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
39
40     if (f == MAP_FAILED)
41         throwError("Error: Failed to mmap", NULL);
42
43     fifo_init(f);
44
45     // MAKE WRITER PORCESSES
46     for (i = 0; i < WRITERS; i++)
47     {
48         if ((pid_table[i] = fork()) < 0)
49             throwError("Error: Failed to fork process.", "Writer");
50     }
```



```
47
48     else if (pid_table[i] == 0)
49     {
50         pid_table[i] = getpid();
51         procNum = i;
52
53         for (j = 0; j < NUMITR; j++)
54         {
55             datum = pid_table[i] * 10000 + j;
56             fifo_wr(f, datum);
57             printf("WRITE %lu by PID: %d\n", datum, pid_table[i]);
58             fprintf(writes, "%lu\n", datum);
59         }
60         return 0;
61     }
62 }
63
64 // MAKE SINGLE READER PROCESS
65 if ((pid_table[WRITERS] = fork()) < 0)
66     throwError("Error: Failed to fork process.", "Reader");
67 else if (pid_table[WRITERS] == 0)
68 {
69     pid_table[WRITERS] = getpid();
70     procNum = WRITERS;
71
72     for (i = 0; i < (WRITERS * NUMITR); i++)
73     {
74         datum = fifo_rd(f);
75         printf("READ %lu by PID: %d\n", datum, pid_table[WRITERS]);
76         fprintf(reads, "%lu\n", datum);
77     }
78     return 0;
79 }
80
81 for (i = 0; i < (WRITERS + 1); i++)
82 {
83     if (waitpid(pid_table[i], NULL, 0) < 0)
84         throwError("Error: Unable to wait for child process to complete",
85 NULL);
86 }
87 return 0;
88 }
```