Dhvanil Shah
ECE 357
Prof. Hakner
14 December 2018

Problem Set 6

Table of Contents

Write Up **1**

Code

Makefile *4*

tas64.S (MacOS) *5*

spin.c *6*

spin.h *7*

sem.c *8*

sem.h *10*

fifo.c *11*

fifo.h *13*

testTAS.c *14*

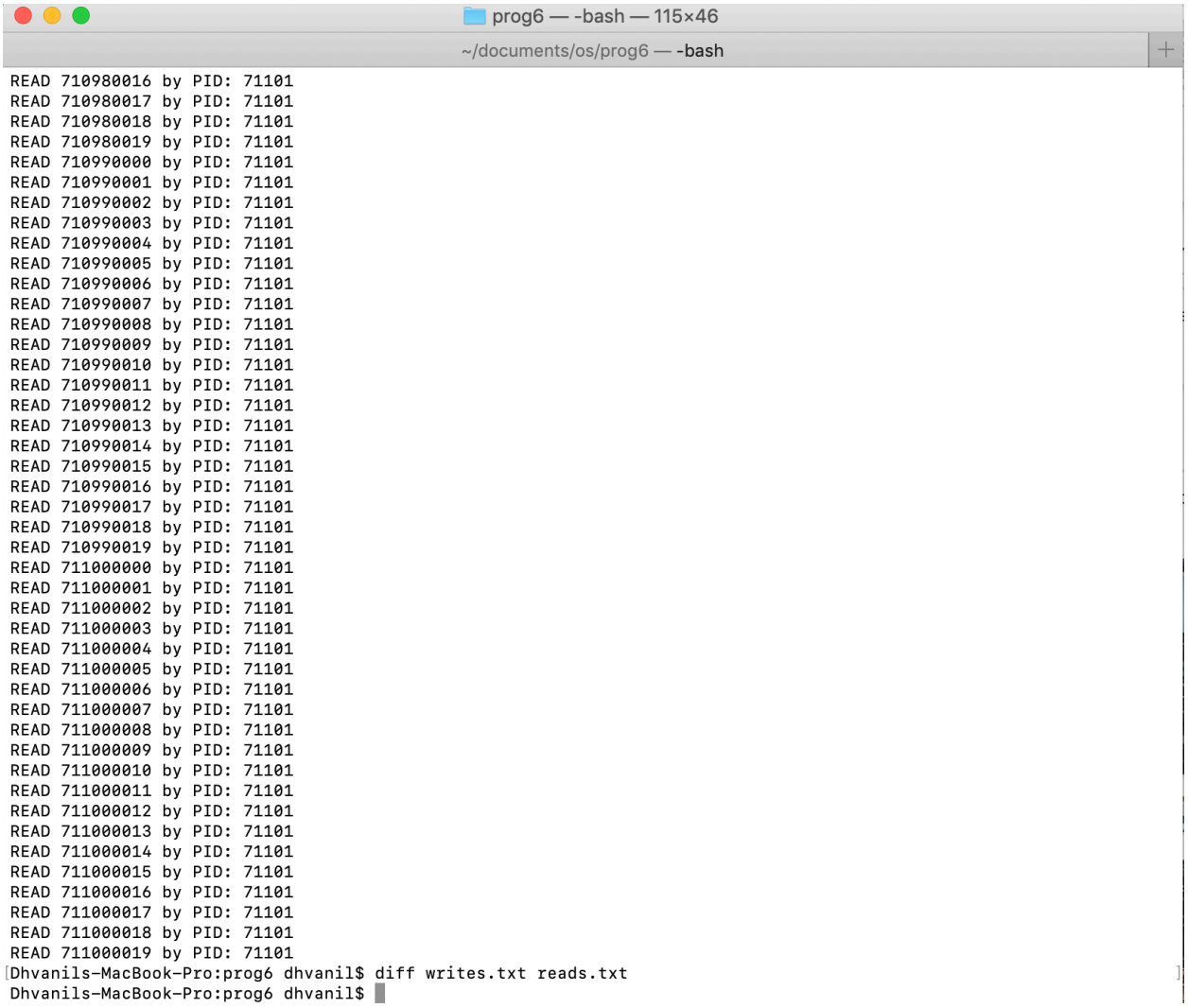
testFIFO.c *16*

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.



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
```

```
.PHONY: clean

testFIFO: testFIFO.o tas64.o sem.o fifo.o spin.o
  @echo "Building 'testTAS'..."
  @gcc -o testFIFO testFIFO.o tas64.o sem.o fifo.o spin.o

testTAS: testTAS.o tas64.o spin.o
  @echo "Building 'testTAS'..."
  @gcc -o testTAS testTAS.o tas64.o spin.o

testTAS.o: testTAS.c
  @echo "Building 'testTAS.o'..."
  @gcc -c testTAS.c

spin.o: spin.c spin.h
  @echo "Building 'spin.o'..."
  @gcc -c spin.c

sem.o: sem.c sem.h
  @echo "Building 'sem.o'..."
  @gcc -c sem.c

fifo.o: fifo.c fifo.h
  @echo "Building 'fifo.o'..."
  @gcc -c fifo.c

testFIFO.o: testFIFO.c
  @echo "Building 'testFIFO.o'..."
  @gcc -c testFIFO.c

tas64.o: tas64.S
  @echo "Building 'tas.o'..."
  @gcc -c tas64.S


clean:
  @echo "Removing all built files..."
  @rm -f *.o testTAS testFIFO
  @clear
```

```
  .text
.globl _tas
_tas:
  pushq %rbp
  movq  %rsp, %rbp
  movq  $1, %rax
#APP
  lock;xchgb  %al,(%rdi)
#NO_APP
  movsbq  %al,%rax
  pop %rbp
  ret
Lfe1:
```

```c
#include "spin.h"
#include "tas.h"
#include <sched.h>

void spin_lock(volatile char *lock)
{
    while (tas(lock))
        sched_yield(); // No need to check sched yeild because it always succeeds in the linux implementation
}

void spin_unlock(volatile char *lock)
{
    *lock = 0;
}
```

```
#ifndef __SPIN_H
void spin_lock(volatile char *lock);
void spin_unlock(volatile char *lock);
#define __SPIN_H
#endif
```

```c
#include "sem.h"
#include "spin.h"

static void dummy() //Dummy handler. Does nothing
{
    // return;
}

void sem_init(struct sem *s, int count)
{
    s->spinlock = 0;
    s->max = count;
    s->free = count;
    s->procInd = -1;

    sigfillset(&s->mask_block);
    sigdelset(&s->mask_block, SIGUSR1);
    signal(SIGUSR1, dummy); // Prevent the signal from killing the process
}

int sem_try(struct sem *s)
{
    spin_lock(&s->spinlock);
    if (s->free > 0)
    {
        s->free -= 1;
        spin_unlock(&s->spinlock);
        return 1;
    }
    else
    {
        spin_unlock(&s->spinlock);
        return 0;
    }
}

void sem_wait(struct sem *s)
{
    for (;;)
    {
        spin_lock(&s->spinlock);

        if (s->free > 0)
        {
            s->free -= 1;
            spin_unlock(&s->spinlock);
            break;
        }
        else
        {
```

```
        {
            s->proc_block[s->procInd] = procNum; // Put process on waitlist
            s->procInd += 1;                     //Book keeping
            spin_unlock(&s->spinlock);
            sigsuspend(&s->mask_block); //Put process to sleep
        }
    }
}

void sem_inc(struct sem *s)
{
    spin_lock(&s->spinlock);

    s->free += 1; // Increment semaphore
    if (s->free == 1)
    {
        while (s->procInd != -1) //Loop to wake up all processes when sem
becomes 1
        {
            kill(pid_table[s->proc_block[s->procInd]], SIGUSR1);
            s->procInd -= 1; // Book keeping
        }
    }
    spin_unlock(&s->spinlock);
}
```

```
#ifndef __SEM_H

#include <signal.h>

#define NUM_PROC 64
#define MYPROCS 4

extern int procNum;      // Current Process Index
extern pid_t *pid_table; // Table of Process PIDS

struct sem
{
    char spinlock;            // The Lock
    int max;                  // Max Resources Available
    int free;                 // Actaul Available Resources
    int procInd;              // Index of proc_block for book keeping
    int proc_block[NUM_PROC]; //List of Blocking Processes
    sigset_t mask_block;      //Mask for all signals but SIGUSR1
};

//   Initialize the semaphore *s with the initial count. Initialize
//   any underlying data structures.  sem_init should only be called
//   once in the program (per semaphore).  If called after the
//   semaphore has been used, results are unpredictable.
void sem_init(struct sem *s, int count);

//   Attempt to perform the "P" operation (atomically decrement
//   the semaphore).  If this operation would block, return 0,
//   otherwise return 1.
int sem_try(struct sem *s);

//   Perform the P operation, blocking until successful.
void sem_wait(struct sem *s);

//   Perform the V operation.  If any other tasks were sleeping
//   on this semaphore, wake them by sending a SIGUSR1 to their
//   process id (which is not the same as the virtual processor number).
//   If there are multiple sleepers (this would happen if multiple
//   virtual processors attempt the P operation while the count is <1)
//   then \fBall\fP must be sent the wakeup signal.
void sem_inc(struct sem *s);
#define __SEM_H
#endif
```

```c
#include "fifo.h"

void fifo_init(struct fifo *f)
{
    f->next_read = 0;
    f->next_write = 0;
    sem_init(&f->empty, MYFIFO_BUFSIZ);
    sem_init(&f->full, 0);
    sem_init(&f->mutex, 1);
}

void fifo_wr(struct fifo *f, unsigned long d)
{
    for (;;)
    {
        sem_wait(&f->empty);

        if (sem_try(&f->mutex))
        {
            f->buffer[f->next_write] = d;
            f->next_write = (f->next_write + 1) % MYFIFO_BUFSIZ;
            sem_inc(&f->mutex);
            sem_inc(&f->full);
            break;
        }
        else
        {
            sem_inc(&f->empty);
        }
    }
}

unsigned long fifo_rd(struct fifo *f)
{
    unsigned long d;
    for (;;)
    {
        sem_wait(&f->full);
        if (sem_try(&f->mutex))
        {
            d = f->buffer[f->next_read];
            f->next_read = (f->next_read + 1) % MYFIFO_BUFSIZ;
            sem_inc(&f->mutex);
            sem_inc(&f->empty);
            break;
        }
        else
        {
            sem_inc(&f->full);
        }
```

```
        }
    }
    return d;
}
```

```
#ifndef __FIFO_H
#include "sem.h"

#define MYFIFO_BUFSIZ 4096

struct fifo
{
    unsigned long buffer[MYFIFO_BUFSIZ];
    int next_read, next_write;
    struct sem empty; // empty spots
    struct sem full;  // filled spots
    struct sem mutex; // mutex for operations
};

//    Initialize the shared memory FIFO *f including any
//    required underlying initializations (such as calling sem_init)
//    The FIFO will have a fifo length of MYFIFO_BUFSIZ elements,
//    which should be a static #define in fifo.h (a value of 4K is
//    reasonable).
void fifo_init(struct fifo *f);

//    Enqueue the data word d into the FIFO, blocking
//    unless and until the FIFO has room to accept it.
//    Use the semaphore primitives to accomplish blocking and waking.
//    Writing to the FIFO shall cause any and all processes that
//    had been blocked because it was empty to wake up.
void fifo_wr(struct fifo *f, unsigned long d);

//    Dequeue the next data word from the FIFO and return it.
//    Block unless and until there are available words
//    queued in the FIFO.  Reading from the FIFO shall cause
//    any and all processes that had been blocked because it was
//    full to wake up.
unsigned long fifo_rd(struct fifo *f);

#define __FIFO_H
#endif
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <errno.h>
#include <string.h>
#include <ctype.h>

#include "spin.h"

#define NUM_PROC 64
#define MYPROCS 4
#define NUMITR 1000000

void throwError(char *message, char *file)
{
    if (file)
        fprintf(stderr, "%s [%s]: Error code %i: %s\n", message, file, errno, strerror(errno));
    else
        fprintf(stderr, "%s\n", message);
    exit(-1);
}

int main(int argc, char const *argv[])
{
    int pid[MYPROCS], myPID = 0;

    unsigned long long idealCt = (MYPROCS * NUMITR);
    unsigned long long *counter = (unsigned long long *)mmap(NULL, sizeof(unsigned long long), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    unsigned long long *counterTAS = (unsigned long long *)mmap(NULL, sizeof(unsigned long long), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    char *lock = (char *)mmap(NULL, sizeof(char), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    for (int i = 0; i < MYPROCS; i++)
    {
        if ((pid[i] = fork()) < 0)
        {
            throwError("Error: Failed to fork process.", NULL);
        }
        else if (pid[i] == 0)
        {
            myPID = 0;
```

```
            for (i = 0; i < NUMITR; i++)
            {
                *counter += 1;
            }

            for (i = 0; i < NUMITR; i++)
            {
                spin_lock(lock);
                *counterTAS += 1;
                spin_unlock(lock);
            }
            break;
        }
        else
            myPID = 1;
    }

    if (myPID)
    {
        for (int i = 0; i < MYPROCS; i++)
        {
            if (waitpid(pid[i], NULL, 0) < 0)
            {
                throwError("Error: Unable to wait for child process to complete", NULL);
            }
        }
        // PRINT OUT RESULTS
        fprintf(stderr, "IDEAL COUNT: %llu | NON-TAS COUNT: %llu | TAS COUNT: %llu\n", idealCt, *counter, *counterTAS);

        if ((munmap(counter, sizeof(unsigned long long)) < 0) || (munmap(counterTAS, sizeof(unsigned long long)) < 0))
            throwError("Error: Unable to munmap counter(s)", 0);

        if ((munmap(lock, sizeof(char)) < 0))
            throwError("Error: Unable to munmap lock", 0);
    }

    return 0;
}

```

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <errno.h>
#include <string.h>
#include "fifo.h"

#define WRITERS 5
#define NUMITR 200

int procNum;
pid_t *pid_table;

void throwError(char *message, char *file)
{
    if (file)
        fprintf(stderr, "%s [%s]: Error code %i: %s\n", message, file, errno, strerror(errno));
    else
        fprintf(stderr, "%s\n", message);
    exit(-1);
}

int main(int argc, char **argv)
{
    struct fifo *f;
    int i, j;
    unsigned long datum;
    FILE *writes = fopen("writes.txt", "w"), *reads = fopen("reads.txt", "w");
    if (writes == NULL || reads == NULL)
        throwError("Error: Unable to open reads and writes log", NULL);

    f = (struct fifo *)mmap(NULL, sizeof(struct fifo), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    pid_table = (pid_t *)mmap(NULL, ((sizeof(pid_t)) * NUM_PROC), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    if (f == MAP_FAILED)
        throwError("Error: Failed to mmap", NULL);

    fifo_init(f);

    // MAKE WRITER PORCESSES
    for (i = 0; i < WRITERS; i++)
    {
        if ((pid_table[i] = fork()) < 0)
            throwError("Error: Failed to fork process.", "Writer");
```

```
        else if (pid_table[i] == 0)
        {
            pid_table[i] = getpid();
            procNum = i;

            for (j = 0; j < NUMITR; j++)
            {
                datum = pid_table[i] * 10000 + j;
                fifo_wr(f, datum);
                printf("WRITE %lu by PID: %d\n", datum, pid_table[i]);
                fprintf(writes, "%lu\n", datum);
            }
            return 0;
        }
    }

    // MAKE SINGLE READER PROCESS
    if ((pid_table[WRITERS] = fork()) < 0)
        throwError("Error: Failed to fork process.", "Reader");
    else if (pid_table[WRITERS] == 0)
    {
        pid_table[WRITERS] = getpid();
        procNum = WRITERS;

        for (i = 0; i < (WRITERS * NUMITR); i++)
        {
            datum = fifo_rd(f);
            printf("READ %lu by PID: %d\n", datum, pid_table[WRITERS]);
            fprintf(reads, "%lu\n", datum);
        }
        return 0;
    }

    for (i = 0; i < (WRITERS + 1); i++)
    {
        if (waitpid(pid_table[i], NULL, 0) < 0)
            throwError("Error: Unable to wait for child process to complete",
NULL);
    }
    return 0;
}
```