Stanley George
Professor Jeff Hakner
ECE 460: Computer Operating Systems
Problem Set 6: Synchronization and Semaphores

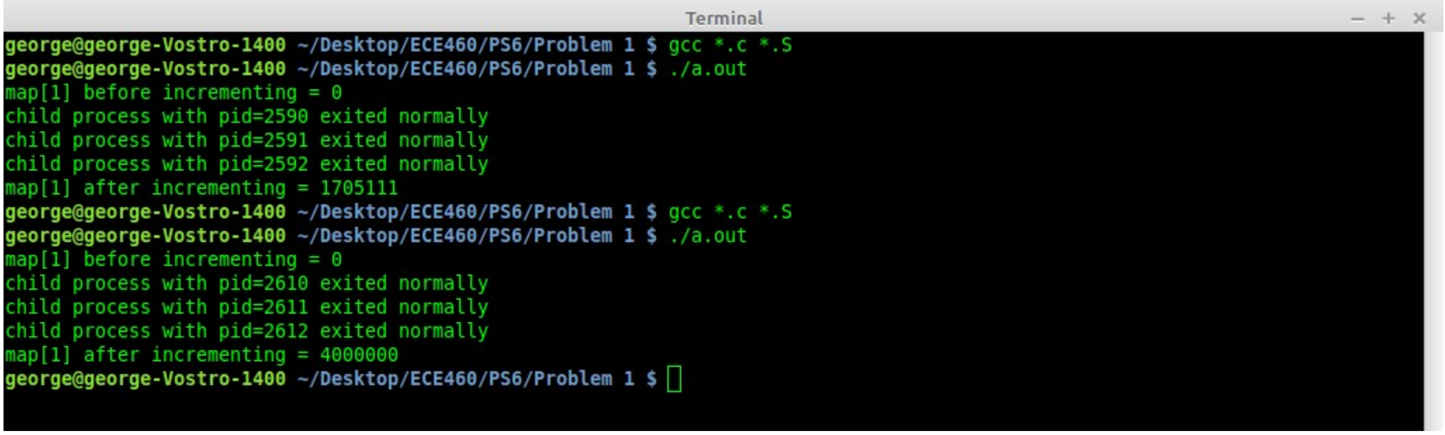
Problem 1)

The code below was used to increment a variable in shared memory between 4 processes (parent and three children).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/uio.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <signal.h>

#define MAP_SIZE 4096
#define UNLOCKED 0
#define LOCKED   1

static long int *map;
static long int i;
int my_procnum;
extern int tas(volatile char *lock);
int acquire_lock(volatile char *lock);
void release_lock(volatile char *lock);
int waitchild(pid_t pid, int *status);

int main(int argc, char *argv[]) {
  pid_t pid1, pid2, pid3;              /*spawn 3 processes because we have 2 cores*/
  int status1, status2, status3;
  if((map = (long int *) mmap((void *) NULL, MAP_SIZE, PROT_READ|PROT_WRITE,
    MAP_ANONYMOUS | MAP_SHARED, -1, 0)) == MAP_FAILED) { perror("mmap"); exit(1); }

  printf("map[1] before incrementing = %ld\n", map[0]);
  if((pid1 = fork()) == -1) { perror("fork failed for pid1!"); exit(1); }
  if(pid1 > 0)
    if((pid2 = fork()) == -1) { perror("fork failed for pid2!"); exit(1); }
  if(pid1 > 0 && pid2 > 0)
    if((pid3 = fork()) == -1) { perror("fork failed for pid2!"); exit(1); }

  for(i = 0; i < 1000000; i++)
    if((acquire_lock((char )(&map[0]))) == UNLOCKED) { /*comment out this line ...*/
      map[1]++;                        /*critical region*/
    release_lock((char *)(&map[0]));   /*and this line to disable mutex protection*/
    }

  if(pid1 > 0 && pid2 > 0 && pid3 > 0) {
    waitchild(pid1, &status1);
    waitchild(pid2, &status2);
    waitchild(pid3, &status3);
    printf("map[1] after incrementing = %ld\n", map[1]);
  }
  return 0;
}
```

```
int acquire_lock(volatile char *lock)
{ while(tas(lock) == LOCKED) ; return UNLOCKED; }


void release_lock(volatile char *lock)
{ *lock = UNLOCKED; }
int waitchild(pid_t pid, int *status) {
  pid = waitpid(pid, status, 0);
  if(pid == -1) {
    fprintf(stderr, "error waiting for pid=%d %s\n", (int) pid, strerror(errno));
    exit(1);
  }
  if(*status != 0) {
    if(WIFSIGNALED(*status)) {
      fprintf(stderr, "child process with pid=%d exited with signal %d",
        (int) pid, WTERMSIG(*status));
    }
    else
      fprintf(stderr, "child process with pid=%d exited with nonzero value %d\n",
        (int) pid, WEXITSTATUS(*status));
  }
  else
    fprintf(stderr, "child process with pid=%d exited normally\n", (int) pid);
}
```

The results of running the above code without commenting out the indicated lines (the first run of a.out) show that due to the lack of synchronization, the final value of the variable is incorrect. However, using a mutex lock gives the correct results. The results of the above code using mutex locks are shown in the second run of a.out. Through the use of mutex locks, the correct result is obtained.



```
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 1 $ gcc *.c *.S
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 1 $ ./a.out
map[1] before incrementing = 0
child process with pid=2590 exited normally
child process with pid=2591 exited normally
child process with pid=2592 exited normally
map[1] after incrementing = 1705111
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 1 $ gcc *.c *.S
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 1 $ ./a.out
map[1] before incrementing = 0
child process with pid=2610 exited normally
child process with pid=2611 exited normally
child process with pid=2612 exited normally
map[1] after incrementing = 4000000
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 1 $
```

Problem 2)

The code below shows the header files sem.h and queue.h and the .c files sem.c and queue.c which were used to build the semaphore module:

sem.h:* **

```
#ifndef _SEM_H
#define _SEM_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/uio.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <signal.h>

#define N_PROC   64
#define LOCKED   1
#define UNLOCKED 0
#define SEM_BUFSIZE 3

#include "queue.h"

extern int my_procnum;

struct sem {
  char sem_lck;
  int  sem_cnt;
  struct queue sem_tskq;
  struct queue sem_pidq;
};

/*
 * TAS spin lock functions
 */
extern int tas(volatile char *lock);
int acquire_lock(volatile char *lock);
void release_lock(volatile char *lock);

/*
 * semaphore functions
 */
void sem_init(struct sem *s, int count);
int sem_try(struct sem *s);
void sem_wait(struct sem *s);
void sem_inc(struct sem *s);
void sem_sighandler(int signum);

#endif // _SEM_H
```

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


int acquire_lock(volatile char *lock)
{ while(tas(lock) == LOCKED) ; return UNLOCKED; }


void release_lock(volatile char *lock)
{ *lock = UNLOCKED; }


void sem_init(struct sem *s, int count)
{
  init_queue(&(s->sem_tskq));
  init_queue(&(s->sem_pidq));
  s->sem_cnt = count;
  s->sem_lck = UNLOCKED;
  signal(SIGUSR1, sem_sighandler);
}

int sem_try(struct sem *s)
{
  sigset_t oldmask, newmask;
  sigfillset(&newmask);
  sigprocmask(SIG_BLOCK, &newmask, &oldmask);
  if(acquire_lock(&(s->sem_lck)) == UNLOCKED)
  {
    int retval;
    if(s->sem_cnt > 0)
    {
      s->sem_cnt -= 1;
      retval = 1;
    }
    else
      retval = 0;
    release_lock(&(s->sem_lck));
    sigprocmask(SIG_SETMASK, &oldmask, NULL);
    return retval;
  }
}

void sem_wait(struct sem *s)
{
  sigset_t oldmask, newmask;
  sigfillset(&newmask);
  sigprocmask(SIG_BLOCK, &newmask, &oldmask);
  while(1)
  {
    if(acquire_lock(&(s->sem_lck)) == UNLOCKED)
    {
      if(s->sem_cnt > 0)
      {
        s->sem_cnt -= 1;
        sigprocmask(SIG_BLOCK, &oldmask, NULL);
        release_lock(&(s->sem_lck));
        break;
      }
      else
      {
        s->sem_tskq.push(&(s->sem_tskq), my_procnum);
        s->sem_pidq.push(&(s->sem_pidq), getpid());
        sigset_t proc_sigmask;
        sigfillset(&proc_sigmask);
        sigdelset(&proc_sigmask, SIGUSR1);
        release_lock(&(s->sem_lck));
        sigsuspend(&proc_sigmask);
      }
    }
  }
}

void sem_inc(struct sem *s)
{
  sigset_t oldmask, newmask;
  sigfillset(&newmask);
  sigprocmask(SIG_BLOCK, &newmask, &oldmask);
  if(acquire_lock(&(s->sem_lck)) == UNLOCKED)
  {
    s->sem_cnt += 1;
    /*while there are blocked processes, wake them up*/
    while(s->sem_tskq.filled != 0)
    {
      int vid = s->sem_tskq.pop(&(s->sem_tskq));
      pid_t pid = s->sem_pidq.pop(&(s->sem_pidq));
      kill(pid, SIGUSR1);
    }
    sigprocmask(SIG_BLOCK, &oldmask, NULL);
    release_lock(&(s->sem_lck));
  }
}

void sem_sighandler(int signum)
{
  if(signum == SIGUSR1) ;
}
```

queue.h

```
#ifndef QUEUE_H
#define QUEUE_H

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

#define MAX_Q_SIZE 64

struct queue {
  int q_array[MAX_Q_SIZE];
  int head;
  int tail;
  int filled;

  void (*push)(struct queue *q, int data);
  int (*pop)(struct queue *q);
  int (*is_empty)(struct queue *q);
  int (*is_full)(struct queue *q);
};

int init_queue(struct queue *q);
void _push(struct queue *q, int data);
int _pop(struct queue *q);
int _is_empty(struct queue *q);
int _is_full(struct queue *q);

#endif //QUEUE_H
```

queue.c

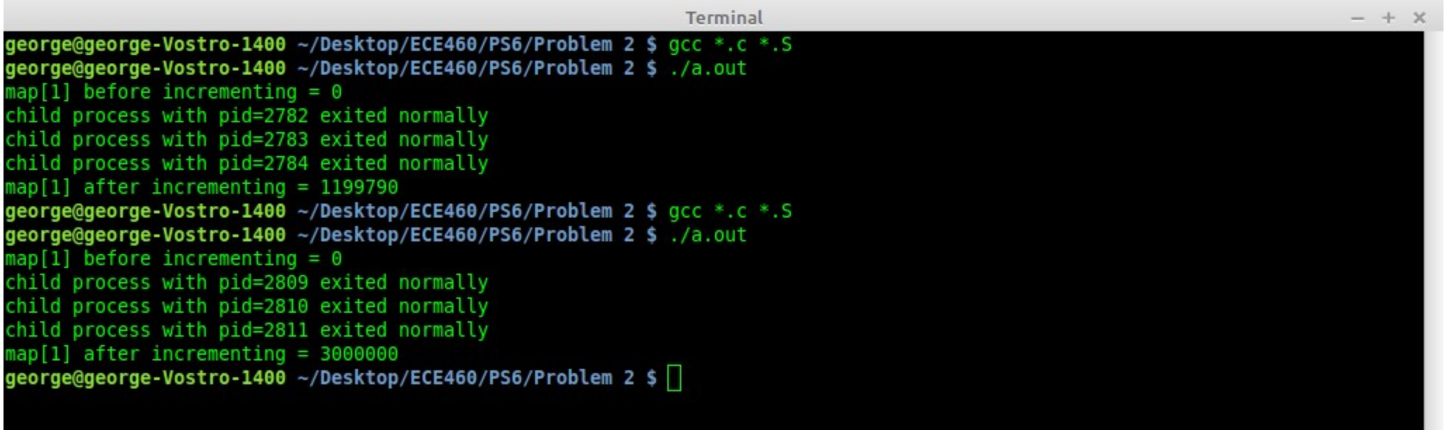
```
#include "queue.h"

int init_queue(struct queue *q)
{
  q->push = &_push;
  q->pop = &_pop;
  q->is_empty = &_is_empty;
  q->is_full = &_is_full;

  q->head = q->tail = -1;
  q->filled = 0;
}

void _push(struct queue *q, int data)
{
  q->tail++;
  q->q_array[q->tail % MAX_Q_SIZE] = data;
  q->filled++;
}

int _pop(struct queue *q)
{
  q->head++;
  int retval = q->q_array[q->head % MAX_Q_SIZE];
  q->filled--;
  return retval;
}

int _is_empty(struct queue *q)
{
  return (q->filled == 0);
}

int _is_full(struct queue *q)
{
  return (q->filled == MAX_Q_SIZE);
}
```

The code below was used to test the functionality of the semaphore module developed above:

semtest.c

```
#include "sem.h"

#define NUM_CHILD 3
#define MAP_SIZE 4096

int my_procnum;
int cpid_list[NUM_CHILD];
int cpid_status[NUM_CHILD];
long int *map;

void spawn_proc(struct sem *s, int nproc);
void pchildstatus(int status, int pid);

int main() {
  struct sem *s;
  if((s = (struct sem *) mmap(0, sizeof(struct sem), PROT_READ|PROT_WRITE,
    MAP_SHARED|MAP_ANONYMOUS, -1, 0)) == MAP_FAILED) { perror("map"); exit(1); }
  sem_init(s, 1);    /*sem count = 1 -> mutex lock*/

  if((map = (long int *) mmap((void *) NULL, MAP_SIZE, PROT_READ|PROT_WRITE,
    MAP_ANONYMOUS | MAP_SHARED, -1, 0)) == MAP_FAILED) { perror("mmap"); exit(1); }

  printf("map[1] before incrementing = %ld\n", map[1]);
  spawn_proc(s, NUM_CHILD);
  int i;
  for(i = 0; i < NUM_CHILD; i++)
    pchildstatus(cpid_status[i], cpid_list[i]);

  printf("map[1] after incrementing = %ld\n", map[1]);
  return 0;
}

void spawn_proc(struct sem *s, int nproc){
  int cpid, status;
  long int i;

  for (my_procnum = 0; my_procnum < nproc; my_procnum++) {
    if((cpid = fork()) == -1) { perror("fork"); exit(1); }
    if(cpid == 0) {
      for (i = 0; i < 1000000; i++) {
        sem_wait(s); /*attempt to obtain semaphore, otherwise block till success*/
        map[1] += 1; /*critical region*/
        sem_inc(s);  /*release semaphore to other processes*/
      }
      exit(0);
    }
    else {
      cpid_list[my_procnum] = cpid;
    }
  }

  i = 0;
  while(1) {
    wait(&status);
    cpid_status[i] = status;
    if(errno == ECHILD) break;
    i++;
  }
}

void pchildstatus(int status, int pid) {
  int i;
  if(status != 0) {
    if(WIFSIGNALED(status)) {
      fprintf(stderr, "child process with pid=%d exited with signal %d",
        pid, WTERMSIG(status));
    }
    else
      fprintf(stderr, "child process with pid=%d exited with nonzero value %d\n",
        pid, WEXITSTATUS(status));
  }
  else
    fprintf(stderr, "child process with pid=%d exited normally\n", pid);
}
```

When the lines right before and right after the critical region (as indicated by the comments) were commented out, the results shown were given by the first run of a.out in the following screenshot:



```
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 2 $ gcc *.c *.S
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 2 $ ./a.out
map[1] before incrementing = 0
child process with pid=2782 exited normally
child process with pid=2783 exited normally
child process with pid=2784 exited normally
map[1] after incrementing = 1199790
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 2 $ gcc *.c *.S
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 2 $ ./a.out
map[1] before incrementing = 0
child process with pid=2809 exited normally
child process with pid=2810 exited normally
child process with pid=2811 exited normally
map[1] after incrementing = 3000000
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 2 $ 
```

The results show that when 3 children increment map[1] for a million iterations, the final number is erroneous as it doesn't reflect the increment of map[1] by each child process. However, with the addition of semaphores (the second run of a.out) the correct result of 3 million is observed.

Problem 3)

The code below shows the header file fifo.h and the .c file fifo.c used to build the fifo module:

fifo.h*

```
#ifndef FIFO_H
#define FIFO_H

#define MYFIFO_BUFSIZ 4096

struct fifo {
  struct sem *sem_rd;
  struct sem *sem_wr;
  struct sem *sem_fifo;

  int head;
  int tail;
  int filled;
  unsigned long buf[MYFIFO_BUFSIZ];

  void (*fifo_push)(struct fifo *f, unsigned long data);
  unsigned long (*fifo_pop)(struct fifo *f);
  int (*fifo_is_empty)(struct fifo *f);
  int (*fifo_is_full)(struct fifo *f);
};

void fifo_init(struct fifo *f);
void fifo_wr(struct fifo *f, unsigned long d);
unsigned long fifo_rd(struct fifo *f);

void _fifo_push(struct fifo *f, unsigned long data);
unsigned long _fifo_pop(struct fifo *f);
int _fifo_is_empty(struct fifo *f);
int _fifo_is_full(struct fifo *f);

#endif // FIFO_H
```

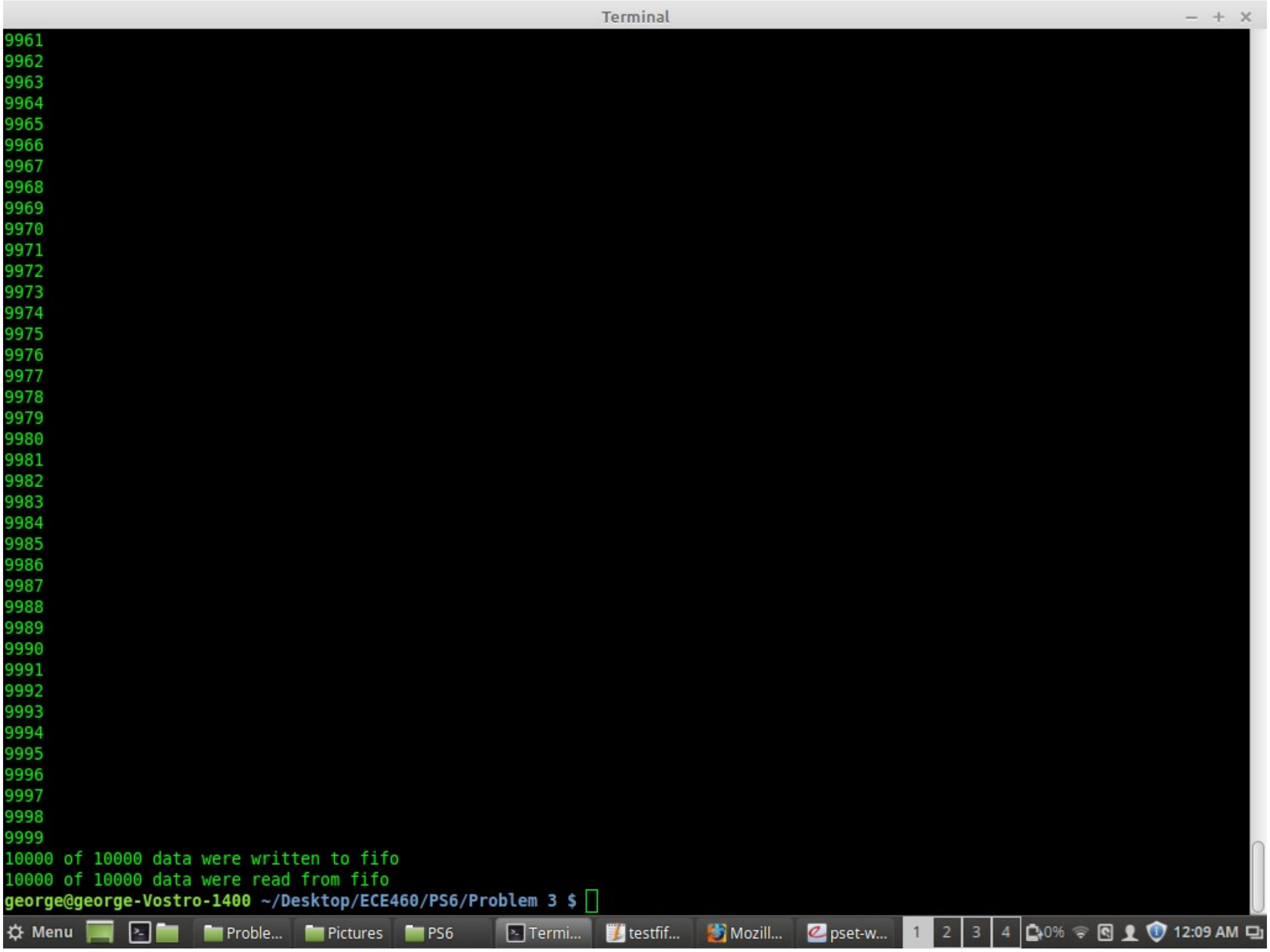
```
***************************************************fifo.c**************************************************
#include "sem.h"
#include "fifo.h"

void fifo_init(struct fifo *f)
{
  if((f->sem_rd = (struct sem *) mmap(0, sizeof(struct sem), PROT_READ|PROT_WRITE,
    MAP_SHARED|MAP_ANONYMOUS, -1, 0)) == MAP_FAILED) { perror("map"); exit(1); }
  if((f->sem_wr = (struct sem *) mmap(0, sizeof(struct sem), PROT_READ|PROT_WRITE,
    MAP_SHARED|MAP_ANONYMOUS, -1, 0)) == MAP_FAILED) { perror("map"); exit(1); }
  if((f->sem_fifo = (struct sem *) mmap(0, sizeof(struct sem), PROT_READ|PROT_WRITE,
    MAP_SHARED|MAP_ANONYMOUS, -1, 0)) == MAP_FAILED) { perror("map"); exit(1); }

  sem_init(f->sem_rd, 0);
  sem_init(f->sem_wr, MYFIFO_BUFSIZ);
  sem_init(f->sem_fifo, 1);

  f->head = f->tail = -1;
  f->filled = 0;

  f->fifo_is_empty = &_fifo_is_empty;
  f->fifo_is_full = &_fifo_is_full;
  f->fifo_push = &_fifo_push;
  f->fifo_pop = &_fifo_pop;
}


void fifo_wr(struct fifo *f, unsigned long data)
{
  sem_wait(f->sem_wr);   /*attempt to obtain write access; sleep till success*/
  sem_wait(f->sem_fifo); /*attempt to obtain fifo access; sleep till success*/
  f->fifo_push(f, data); /*critical region; only 1 proccess at a time*/
  sem_inc(f->sem_fifo);  /*open fifo access to other processes*/
  sem_inc(f->sem_rd);    /*open read access to other processes*/
}

unsigned long fifo_rd(struct fifo *f)
{
  unsigned long retval;
  sem_wait(f->sem_rd);     /*attempt to obtain read access; sleep till success*/
  sem_wait(f->sem_fifo);   /*attempt to obtain fifo access; sleep till success*/
  retval = f->fifo_pop(f); /*critical region; only 1 proccess at a time*/
  sem_inc(f->sem_fifo);    /*open fifo access to other processes*/
  sem_inc(f->sem_wr);      /*open write access to other processes*/
  return retval;
}


void _fifo_push(struct fifo *f, unsigned long data)
{
  f->tail++;
  f->buf[f->tail % MYFIFO_BUFSIZ] = data;
  f->filled++;
}

unsigned long _fifo_pop(struct fifo *f)
{
  f->head++;
  unsigned long retval = f->buf[f->head % MYFIFO_BUFSIZ];
  f->filled--;
  return retval;
}

int _fifo_is_empty(struct fifo *f)
{
  return (f->filled == 0);
}

int _fifo_is_full(struct fifo *f)
{
  return (f->filled == MYFIFO_BUFSIZ);
}
```

Problem 4a)

The following code was used to perform a simple test in which one process wrote to the fifo and another read from the fifo. MY_FIFO_BUFSIZE was set to 1 for this test so that one packet of data was written to and read from the data at a time. A total of MAX_WRITES = 10000 of such reads and writes were made.

```
#include "sem.h"
#include "fifo.h"

#define MAX_WRITE 10000
#define N_READ    0
#define N_WRITTEN 1

int my_procnum;
unsigned long *write_array;
int cpid_status[2];
int cpid_list[2];
int *data_count;

void pchildstatus(int status, int pid);

int main()
{
  struct fifo *f;
  if((f = (struct fifo *) mmap(0, sizeof(struct fifo), PROT_READ|PROT_WRITE,
    MAP_SHARED|MAP_ANONYMOUS, -1, 0)) == MAP_FAILED) { perror("map"); exit(1); }
  if((write_array = mmap(0, MAX_WRITE * sizeof(unsigned long), PROT_READ|PROT_WRITE,
    MAP_SHARED|MAP_ANONYMOUS, -1, 0)) == MAP_FAILED) { perror("map"); exit(1); }
  if((data_count = mmap(0, 2 * sizeof(int), PROT_READ|PROT_WRITE,
    MAP_SHARED|MAP_ANONYMOUS, -1, 0)) == MAP_FAILED) { perror("map"); exit(1); }

  fifo_init(f);

  pid_t reader, writer;
  for(my_procnum = 0; my_procnum < 1; my_procnum++)  /*spawn reader*/
  {
    if((reader = fork()) == -1) { perror("fork"); exit(1); }
    if(reader == 0)
    {
      int i;
      for(i = 0; i < MAX_WRITE; i++)
      {
        write_array[i] = fifo_rd(f);
        data_count[N_READ] += 1;
      }
      exit(0);
    }
    if(reader > 0) cpid_list[0] = reader;
  }

  for(my_procnum = 1; my_procnum < 2; my_procnum++)  /*spawn writer*/
  {
    if((writer = fork()) == -1) { perror("fork"); exit(1); }
    if(writer == 0)
    {
      int i;
      for(i = 0; i < MAX_WRITE; i++)
      {
        fifo_wr(f, i);
        data_count[N_WRITTEN] += 1;
      }
      exit(0);
    }
    if(writer > 0) cpid_list[1] = writer;
  }

  int status;
  wait(&status);
  cpid_status[0] = status;
  wait(&status);
  cpid_status[1] = status;

  int i;
  for(i = 0; i < 2; i++) pchildstatus(cpid_status[i], cpid_list[i]);
  for(i = 0; i < MAX_WRITE; i++) printf("%ld\n", write_array[i]);
  printf("%d of %d data were written to fifo\n", data_count[N_WRITTEN], MAX_WRITE);
  printf("%d of %d data were read from fifo\n", data_count[N_READ], MAX_WRITE);

  return 0;
}

void pchildstatus(int status, int pid)
{
  int i;
  if(status != 0)
  {
    if(WIFSIGNALED(status))
    {
      fprintf(stderr, "child process with pid=%d exited with signal %d",
        pid, WTERMSIG(status));
    }
    else
      fprintf(stderr, "child process with pid=%d exited with nonzero value %d\n",
        pid, WEXITSTATUS(status));
  }
  else
    fprintf(stderr, "child process with pid=%d exited normally\n", pid);
}
```

The truncated output of the above code is shown in the following screenshot. The program also demonstrates that 10000 “packets” of data were indeed sent and received to and by the fifo.



```
9961
9962
9963
9964
9965
9966
9967
9968
9969
9970
9971
9972
9973
9974
9975
9976
9977
9978
9979
9980
9981
9982
9983
9984
9985
9986
9987
9988
9989
9990
9991
9992
9993
9994
9995
9996
9997
9998
9999
10000 of 10000 data were written to fifo
10000 of 10000 data were read from fifo
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 3 $ 
```

Problem 4b) Acid Test

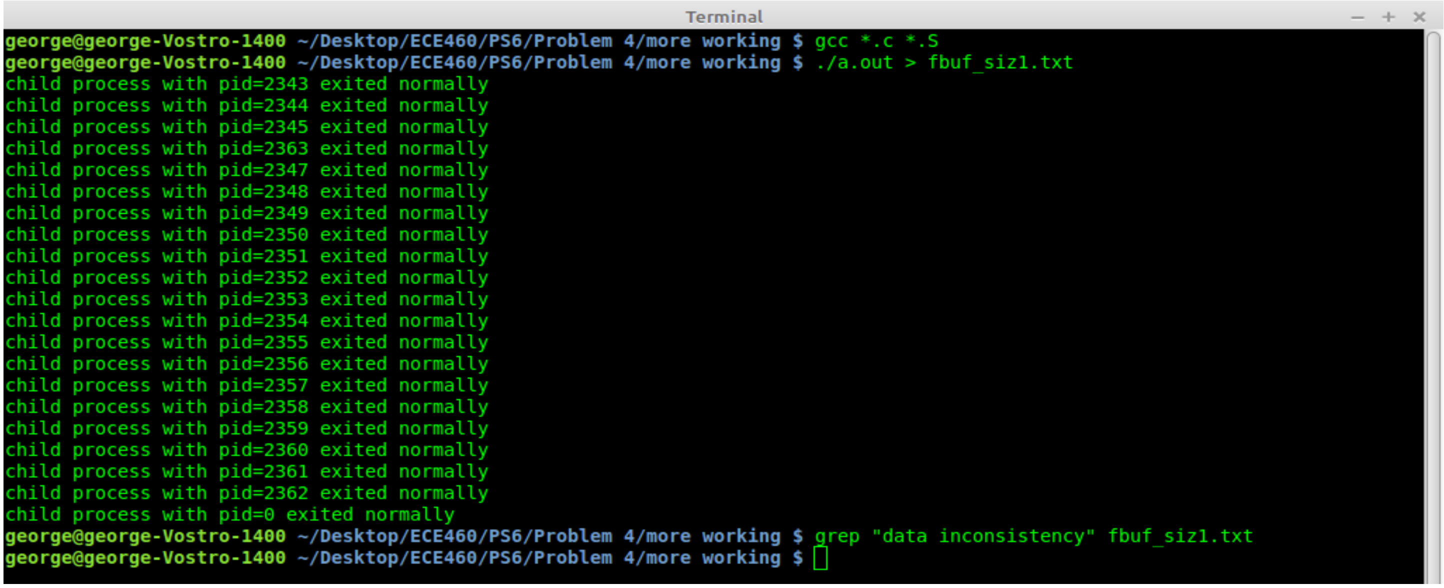
The following code tested the fifo and semaphore modules by spawning 20 writing processes and 1 reader process with MY_FIFO_BUFSIZE = 1 in one run, and with MY_FIFO_BUFSIZE = 100 in another run. In both runs MAX_WRITE = 1000. The following code was called the “acid test”

```
#include "sem.h"
#include "fifo.h"

#define MAX_WRITE 1000              /*each writer shall write MAX_WRITE times to fifo*/
#define N_WRITERS 20                /*there shall be N_WRITERS write processes*/

int my_procnum;                     /*virtual id (vid) of each forked process*/
int cpid_status[N_WRITERS + 1];     /*exit status of each child forked from parent*/
int cpid_array[N_WRITERS + 1];      /*pid of each child forked from parent*/
unsigned long *write_array;         /*stores result of current read operation*/
unsigned long *pwrite_array;        /*stores result of previous read operation*/

void pchildstatus(int status, int pid);

int main()
{
  struct fifo *f;
  int i, j, status;
```

```
if((f = (struct fifo *) mmap(0, sizeof(struct fifo), PROT_READ|PROT_WRITE,
  MAP_SHARED|MAP_ANONYMOUS, -1, 0)) == MAP_FAILED) { perror("map"); exit(1); }


if((write_array = mmap(0, N_WRITERS * sizeof(unsigned long),
  PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0)) == MAP_FAILED)
  { perror("map"); exit(1); }

if((pwrite_array = mmap(0, N_WRITERS * sizeof(unsigned long),
  PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0)) == MAP_FAILED)
  { perror("map"); exit(1); }

pid_t reader, writer;
fifo_init(f);

for(my_procnum = 0; my_procnum < N_WRITERS; my_procnum++)
{
  if((writer = fork()) == -1) { perror("fork"); exit(1); }
  if(writer == 0) {
    int i;
    for(i = 0; i < MAX_WRITE; i++)
    {
      unsigned long seq_num = i;           /*sequence # == current write iteration*/
      unsigned long datum = seq_num << 16;
      datum |= my_procnum;
      fifo_wr(f, datum);                   /*write encoded vid and sequence # to fifo*/
    }
    exit(0);
  }
  else
    cpid_array[my_procnum] = writer;
}

for(my_procnum = N_WRITERS; my_procnum < N_WRITERS + 1; my_procnum++)
{
  if((reader = fork()) == -1) { perror("fork"); exit(1); }
  if(reader == 0){
    int i, j, vid, data;
    unsigned long rdval, mask = 0x0000FFFF;
    for(i = 0; i < N_WRITERS * MAX_WRITE; i++)
    {
      rdval = fifo_rd(f);
      vid = rdval & mask;
      data = rdval >> 16;
      write_array[vid] = data;
      for(j = 0; j < N_WRITERS; j++)
      {
          printf("%ld ", write_array[j]);
      }
      printf("\n");
      for(j = 0; j < N_WRITERS; j++)
      {
          if((pwrite_array[j] + 1 != write_array[j]) &&
            (pwrite_array[j] != write_array[j]))
            printf("data inconsistency!!!!!\n");
          pwrite_array[j] = write_array[j];
      }
    }
    exit(0);
  }
```

```
    else
      cpid_array[my_procnum] = reader;
  }

  while(1) {
    i = 0;
    wait(&status);
    cpid_status[i] = status;
    if(errno == ECHILD) break;
    i++;
      }

  for(i = 0; i < N_WRITERS + 1; i++)
    pchildstatus(cpid_status[i], cpid_array[i]);


  printf("done\n");
  return 0;
}



void pchildstatus(int status, int pid) {
  int i;
  if(status != 0) {
    if(WIFSIGNALED(status)) {
      fprintf(stderr, "child process with pid=%d exited with signal %d",
        pid, WTERMSIG(status));
    }
    else
      fprintf(stderr, "child process with pid=%d exited with nonzero value %d\n",
        pid, WEXITSTATUS(status));
  }
  else
    fprintf(stderr, "child process with pid=%d exited normally\n",
      pid);
}
```

The screenshot below shows the program did not hang or crash when using a bufsize of 1. It should be noted that all the children exited successfully. There was one bug in the program where the pid of the last child would be seen as 0, but this was not a serious error. The intent was just to make sure ALL the children exited successfully.



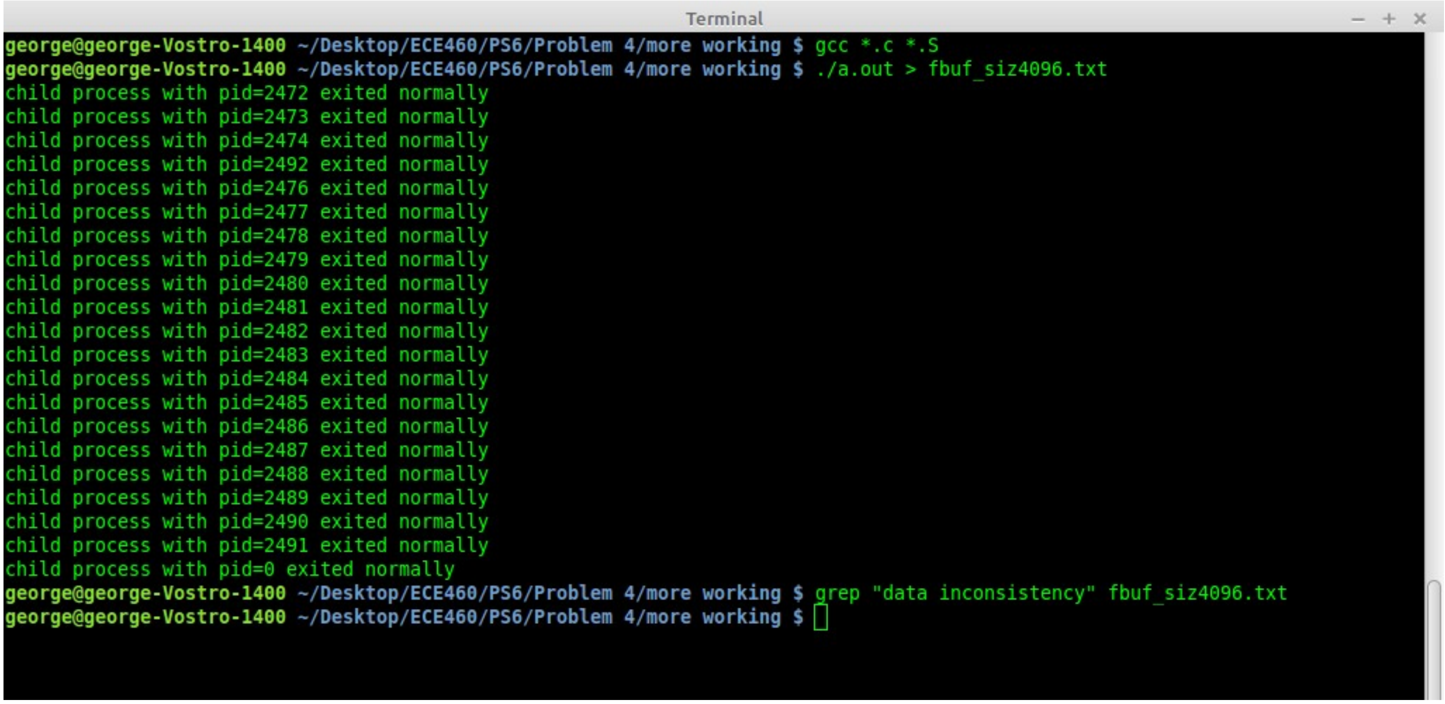
```
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 4/more working $ gcc *.c *.S
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 4/more working $ ./a.out > fbuf_siz1.txt
child process with pid=2343 exited normally
child process with pid=2344 exited normally
child process with pid=2345 exited normally
child process with pid=2363 exited normally
child process with pid=2347 exited normally
child process with pid=2348 exited normally
child process with pid=2349 exited normally
child process with pid=2350 exited normally
child process with pid=2351 exited normally
child process with pid=2352 exited normally
child process with pid=2353 exited normally
child process with pid=2354 exited normally
child process with pid=2355 exited normally
child process with pid=2356 exited normally
child process with pid=2357 exited normally
child process with pid=2358 exited normally
child process with pid=2359 exited normally
child process with pid=2360 exited normally
child process with pid=2361 exited normally
child process with pid=2362 exited normally
child process with pid=0 exited normally
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 4/more working $ grep "data inconsistency" fbuf_siz1.txt
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 4/more working $ 
```

The output of the above run was sent to the file fbuf_siz1.txt. The contents of the file are too large to replicate here but the last several write iterations are shown for the 20 write processes (each row is a write iteration and each column is the virtual processor id (vid) that wrote the data starting at vid = 0 with the first column). Each process was able to complete its write sequence.

```
999 999 999 999 999 999 999 999 999 999 999 999 983 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 984 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 985 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 986 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 987 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 988 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 989 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 990 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 991 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 992 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 993 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 994 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 995 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 996 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 997 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 998 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
done
```

Also, the contents of fbuf_siz1.txt were passed through grep to find any lines matching the string “data inconsistency”. If ANY such string was found, then that would imply that the write processes were interfering with each other causing non sequential writes. No such string was found which means that the fifo was transmitting data without interleaving them with data from other writers.

Likewise, with MY_FIFO_BUFSIZE = 4096, there was no hangup and data from all writers remained intact through the fifo as the following screen shot and file dump from fbuf_siz4096.txt show.



```
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 4/more working $ gcc *.c *.S
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 4/more working $ ./a.out > fbuf_siz4096.txt
child process with pid=2472 exited normally
child process with pid=2473 exited normally
child process with pid=2474 exited normally
child process with pid=2492 exited normally
child process with pid=2476 exited normally
child process with pid=2477 exited normally
child process with pid=2478 exited normally
child process with pid=2479 exited normally
child process with pid=2480 exited normally
child process with pid=2481 exited normally
child process with pid=2482 exited normally
child process with pid=2483 exited normally
child process with pid=2484 exited normally
child process with pid=2485 exited normally
child process with pid=2486 exited normally
child process with pid=2487 exited normally
child process with pid=2488 exited normally
child process with pid=2489 exited normally
child process with pid=2490 exited normally
child process with pid=2491 exited normally
child process with pid=0 exited normally
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 4/more working $ grep "data inconsistency" fbuf_siz4096.txt
george@george-Vostro-1400 ~/Desktop/ECE460/PS6/Problem 4/more working $ 
```

```
999 999 981 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 982 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 983 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 984 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 985 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 986 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 987 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 988 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 989 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 990 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 991 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 992 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 993 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 994 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 995 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 996 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 997 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 998 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
done
```

The importance of having a single semaphore to control access to the fifo itself was demonstrated when the following changes were made:

FROM -

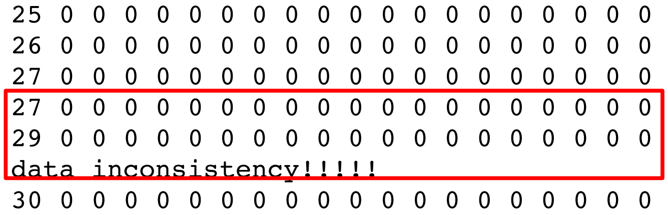
```
void fifo_wr(struct fifo *f, unsigned long data)
{
  sem_wait(f->sem_wr);   /*attempt to obtain write access; sleep till success*/
  sem_wait(f->sem_fifo); /*attempt to obtain fifo access; sleep till success*/
  f->fifo_push(f, data); /*critical region; only 1 proccess at a time*/
  sem_inc(f->sem_fifo);  /*open fifo access to other processes*/
  sem_inc(f->sem_rd);    /*open read access to other processes*/
}

unsigned long fifo_rd(struct fifo *f)
{
  unsigned long retval;
  sem_wait(f->sem_rd);      /*attempt to obtain read access; sleep till success*/
  sem_wait(f->sem_fifo);    /*attempt to obtain fifo access; sleep till success*/
  retval = f->fifo_pop(f); /*critical region; only 1 proccess at a time*/
  sem_inc(f->sem_fifo);     /*open fifo access to other processes*/
  sem_inc(f->sem_wr);       /*open write access to other processes*/
  return retval;
}
```

TO-

```
void fifo_wr(struct fifo *f, unsigned long data)
{
  sem_wait(f->sem_wr);    /*attempt to obtain write access; sleep till success*/
  f->fifo_push(f, data); /*critical region; only 1 proccess at a time*/
  sem_inc(f->sem_rd);     /*open read access to other processes*/
}

unsigned long fifo_rd(struct fifo *f)
{
  unsigned long retval;
  sem_wait(f->sem_rd);      /*attempt to obtain read access; sleep till success*/
  retval = f->fifo_pop(f); /*critical region; only 1 proccess at a time*/
  sem_inc(f->sem_wr);       /*open write access to other processes*/
  return retval;
}
```

When the acid test was performed with the semaphore guarding access to the fifo removed, data from different writers did not remain intact and led to occasional inconsistent writes. MY_FIFO_BUFSIZE for this test was 100. The following shows a snippet of the output of the acid test (this output was written to a file as the output would be too large to fit in a single terminal viewing screen):

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
11 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
12 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
14 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
15 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
17 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
18 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
19 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
21 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
22 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
23 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
24 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```



```
25 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
26 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
27 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
27 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
29 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
data inconsistency!!!!!
30 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

It was also noted that a potential deadlock situation would arise if the order in which semaphores were obtained was changed:

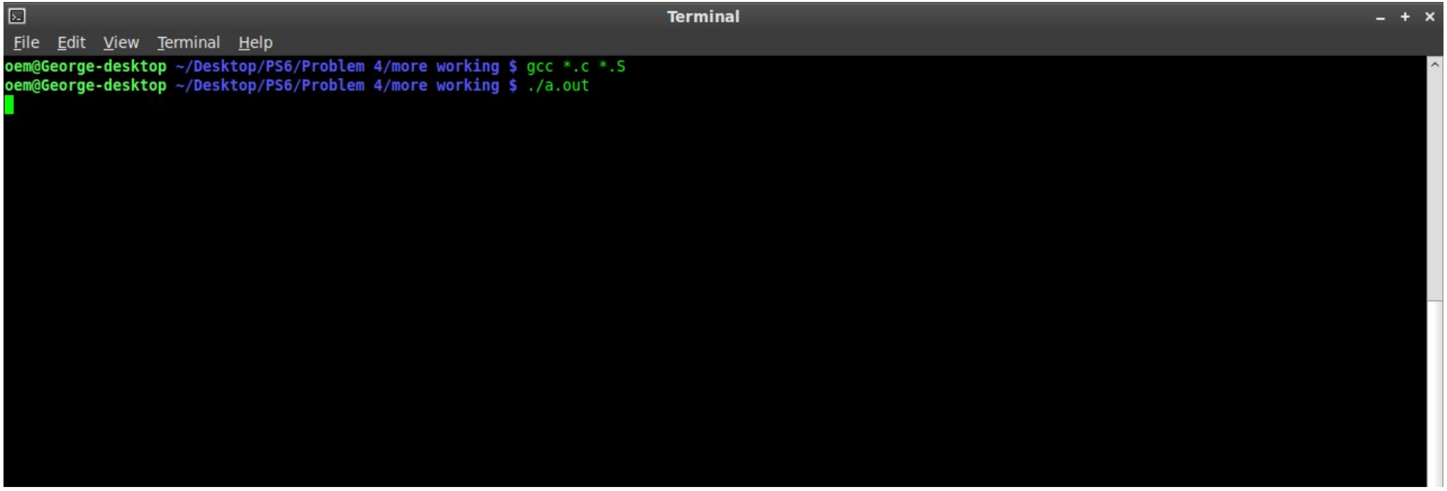
FROM -

```
void fifo_wr(struct fifo *f, unsigned long data)
{
  sem_wait(f->sem_wr);   /*attempt to obtain write access; sleep till success*/
  sem_wait(f->sem_fifo); /*attempt to obtain fifo access; sleep till success*/
  f->fifo_push(f, data); /*critical region; only 1 proccess at a time*/
  sem_inc(f->sem_fifo);  /*open fifo access to other processes*/
  sem_inc(f->sem_rd);    /*open read access to other processes*/
}

unsigned long fifo_rd(struct fifo *f)
{
  unsigned long retval;
  sem_wait(f->sem_rd);     /*attempt to obtain read access; sleep till success*/
  sem_wait(f->sem_fifo);   /*attempt to obtain fifo access; sleep till success*/
  retval = f->fifo_pop(f); /*critical region; only 1 proccess at a time*/
  sem_inc(f->sem_fifo);    /*open fifo access to other processes*/
  sem_inc(f->sem_wr);      /*open write access to other processes*/
  return retval;
}
```

TO -

```
void fifo_wr(struct fifo *f, unsigned long data)
{
  sem_wait(f->sem_fifo); /*attempt to obtain fifo access; sleep till success*/
  sem_wait(f->sem_wr);   /*attempt to obtain write access; sleep till success*/
  f->fifo_push(f, data); /*critical region; only 1 proccess at a time*/
  sem_inc(f->sem_fifo);  /*open fifo access to other processes*/
  sem_inc(f->sem_rd);    /*open read access to other processes*/
}

unsigned long fifo_rd(struct fifo *f)
{
  unsigned long retval;
  sem_wait(f->sem_fifo);   /*attempt to obtain fifo access; sleep till success*/
  sem_wait(f->sem_rd);     /*attempt to obtain read access; sleep till success*/
  retval = f->fifo_pop(f); /*critical region; only 1 proccess at a time*/
  sem_inc(f->sem_fifo);    /*open fifo access to other processes*/
  sem_inc(f->sem_wr);      /*open write access to other processes*/
  return retval;
}
```

The following shows the deadlock situation:



```
oem@George-desktop ~/Desktop/PS6/Problem 4/more working $ gcc *.c *.S
oem@George-desktop ~/Desktop/PS6/Problem 4/more working $ ./a.out
```