Nithilam Subbaian
12/14/2018

Operating Systems: Problem Set 6

Source Code:

spinlock.h:

```
#ifndef __SPINLOCK_H__
#define __SPINLOCK_H__

#include "tas.h"

typedef struct spinlock {
	volatile char p_lock;
}spinlock;

void spin_lock(struct spinlock *lock_arg);

void spin_unlock(struct spinlock *lock_arg);

#endif
```

testSetSpinlock.c:

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

#include "spinlock.h"

int main(int argc, char * argv[]) {

	if(argc!=3) {
		fprintf(stderr,"ERROR:  Specify the number of processes and the number of iterations after
%s\n",argv[0]);
		exit(EXIT_FAILURE);
	}

	long long unsigned int ProcessNumber = atoll(argv[1]), IterationNumber = atoll(argv[2]);

	fprintf (stderr, "Number of Processes = %llu\n", ProcessNumber);
	fprintf (stderr, "Number of Iterations = %llu\n", IterationNumber);

	int * mappedRegion = mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_ANONYMOUS |
MAP_SHARED, 0, 0 );
	int * mappedRegion2 = mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_ANONYMOUS |
MAP_SHARED, 0, 0 );

	if(mappedRegion==MAP_FAILED || mappedRegion2==MAP_FAILED) {
		fprintf(stderr,"ERROR: Failed to mmap ANONYMOUS page(s): %s\n",strerror(errno));
		exit(EXIT_FAILURE);
	}

	mappedRegion[0] = 0;
	mappedRegion2[0] = 0;

	spinlock * lock;
	lock=(spinlock *)(mappedRegion+sizeof(spinlock));
```

```
        lock->p_lock= mappedRegion[1];

        pid_t pids[ProcessNumber];

        for (int i = 0; i < ProcessNumber; i++) {
                if ((pids[i] = fork()) < 0) {
                        fprintf (stderr, "ERROR: Failed to fork for Process Number %d: %s\n", i, strerror (errno));
                        return EXIT_FAILURE;
                }
                if (pids[i] == 0) {
                        for (int jk = 0; jk < IterationNumber; jk++) {
                                mappedRegion2[0]++;
                        }

                        spin_lock(lock);
                        for (int j = 0; j < IterationNumber; j++) {
                                mappedRegion[0]++;
                        }
                        spin_unlock(lock);
                        exit(0);
                }
        }

        for (int ijk = 0; ijk < ProcessNumber; ijk++) {
                if (waitpid (pids[ijk], NULL, 0) < 0) {
                        fprintf (stderr, "ERROR: waitpid failed for reason of: %s\n", strerror (errno));
                }
        }

        printf ("(No. of Processes)*(No. of Iterations):\t%llu\n", ProcessNumber * IterationNumber);
        fprintf(stderr,"With mutex protection:\t\t\t%d\n", mappedRegion[0]);
        fprintf(stderr,"Without mutex protection:\t\t%d\n", mappedRegion2[0]);

}
```

spinlock.c:

```
#include "spinlock.h"

void spin_lock(struct spinlock *lock_arg){
```

```
                                        while(tas(&(lock_arg->p_lock))!=0) ;
}

void spin_unlock(struct spinlock *lock_arg){
                                        lock_arg->p_lock=0;
}
```

testFifo.c:

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

```
#include <sys/wait.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>

#include "fifo.h"

int my_procnum;
pid_t * pid_table;

int main (int argc, char ** argv) {
	struct fifo * f;
	int i, j, numberOfWriters = 2, numberOfIterations = 6;

	unsigned long entry;

	if ((f = (struct fifo *) mmap (NULL, sizeof (struct fifo), PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1, 0))== MAP_FAILED) {
		fprintf (stderr, "ERROR: mmap() failure: %s\n", strerror (errno));
		return -1;
	}

	if ((pid_table = (pid_t *) mmap (NULL, ((sizeof (pid_t)) * N_PROC), PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_ANONYMOUS, -1, 0))== MAP_FAILED) {
		fprintf (stderr, "ERROR: mmap() failure: %s\n", strerror (errno));
		return -1;
	}

	fifo_init (f);

	for (i = 0; i < numberOfWriters; ++i) {
		pid_table[i] = fork ();

		if (pid_table[i] < 0) {
			fprintf (stderr, "ERROR: fork() failure: %s\n", strerror (errno));
			return -1;
		} else if (pid_table[i] == 0) {
			my_procnum = i;
			unsigned long writeBuf[numberOfIterations];
			for (j = 0; j < numberOfIterations; ++j) {
				writeBuf[j] = j + getpid()*10000;
				pid_table[i] = getpid ();
				fifo_wr(f, writeBuf[j]);
				fprintf (stderr, "Process %d wrote %lu to FIFO\n", pid_table[i], writeBuf[j]);
			}
```

```
                fprintf(stderr, "Writer %d completed\n",i);
                return 0;
            }

        }
  fprintf(stderr,"ALL %d Writers done\n", numberOfWriters);
        pid_table[numberOfWriters] = fork ();

        if (pid_table[numberOfWriters] < 0) {
            fprintf (stderr, "ERROR: fork() failure: %s\n", strerror (errno));
            return -1;
        } else if (pid_table[numberOfWriters] == 0) {
            pid_table[numberOfWriters] = getpid ();
            my_procnum = numberOfWriters;

            for (i = 0; i < numberOfWriters * numberOfIterations; ++i) {
                entry = fifo_rd (f);
                fprintf (stderr, "read %lu from FIFO on run %d\n", entry, i);
            }
              fprintf(stderr,"ALL readers done\n");
            return 0;
        }

        for (i = 0; i < (numberOfWriters + 1); ++i) {
            if (waitpid (pid_table[i], NULL, 0) < 0) {
                fprintf (stderr, "ERROR: child process return failure: %s\n", strerror (errno));
                return -1;
            }
        }

        return 0;

}
```

fifo.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);
                                        f->FIFO_lock.p_lock=0;
}

void fifo_wr (struct fifo * f, unsigned long d) {

                                        spin_lock(&f->FIFO_lock);
                                        while (1) {

          sem_wait (&f->empty);

          if (sem_try (&f->mutex)) {

                                                  f->buffer[f->next_write] = d;

                                                  f->next_write++;

                                                  f->next_write %=
MYFIFO_BUFSIZ;

                                                  sem_inc (&f->mutex);

                                                  sem_inc (&f->full);

                                                  break;

          } else {

                                                  sem_inc (&f->empty);

          }
                                        }
                                         spin_unlock(&f->FIFO_lock);
}

unsigned long fifo_rd (struct fifo * f) {
          spin_lock(&f->FIFO_lock);
                                        unsigned long d;
                                        while (1) {

          sem_wait (&f->full);

          if (sem_try (&f->mutex)) {

                                                  d = f->buffer[f->next_read];
```

```
                                                                            f->next_read++;

                                                                            f->next_read %=
MYFIFO_BUFSIZ;

                                                                            sem_inc (&f->mutex);

                                                                            sem_inc (&f->empty);

                                                                            break;

                        } else {

                                                                            sem_inc (&f->full);

                        }
                                                              }
                                                                    spin_unlock(&f->FIFO_lock);
                                                              return d;
}
```

fifo.h:

```
#ifndef __FIFO_H__
#define __FIFO_H__

#include "sem.h"

#define MYFIFO_BUFSIZ 4096

struct fifo {
	unsigned long buffer[MYFIFO_BUFSIZ];
	int next_read;
	int next_write;
```

```
        struct sem empty, full, mutex;
        spinlock FIFO_lock;
};

void fifo_init (struct fifo * f);

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

unsigned long fifo_rd (struct fifo * f);

#endif
```

sem.c:

```
#include "sem.h"

static void handler () {
    //this is a dummy handler for initializations
}

//should be called only once in the program (per semaphore)
void sem_init (struct sem * s, int count) {
    // s->spinlock = 0;
    s->semaphore = count; //initialze the semaphore *s with the initial count

    int * mapped_area = mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_ANONYMOUS |
MAP_SHARED, 0, 0 );
    if(mapped_area==MAP_FAILED) {
            fprintf(stderr,"Failed to mmap ANONYMOUS page[cv_init]: %s\n",strerror(errno));
            exit(EXIT_FAILURE);
    }

    spinlock * lock;
    lock=(spinlock *)(mapped_area+sizeof(spinlock)); /*important:make sure lock is fixed*/
    s->lock=*lock;

    s->prockBlockIndex = -1; // (no blocking processors)
    sigfillset (&s->mask_block);
    sigdelset (&s->mask_block, SIGUSR1); // removes SIGUSR1 from blocked signal list


    if(signal (SIGUSR1, handler)<0) {
            fprintf(stderr,"ERROR: Failed to signal handle: %s\n",strerror(errno));
            exit(EXIT_FAILURE);
    }
}


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

void sem_wait (struct sem * s) {
    //perform the P operation, blocking until successful
    while (1) {
            spin_lock(&s->lock);
            if (s->semaphore > 0) {
                    s->semaphore--;
                    spin_unlock(&s->lock);
                    // printf("FIRSTLOOP");
```

```
            break;
        } else {
            // printf("SECONDLOOP");
            if(sigprocmask(SIG_BLOCK, &s->mask_block, NULL)<0) {
                fprintf(stderr,"ERROR: Failed to examine and change blocked signals:
%s\n",strerror(errno));
                exit(EXIT_FAILURE);
            }
            s->proc_block[s->prockBlockIndex] = my_procnum;
            s->prockBlockIndex++;

            spin_unlock(&s->lock);
            if(sigsuspend (&s->mask_block)<0) {
                fprintf(stderr,"ERROR: Failed to wait for signal: %s\n",strerror(errno));
                exit(EXIT_FAILURE);
            }
            if(sigprocmask(SIG_UNBLOCK, &s->mask_block, NULL)<0) {
                fprintf(stderr,"ERROR: Failed to examine and change blocked signals:
%s\n",strerror(errno));
                exit(EXIT_FAILURE);
            }

        }
    }
}

void sem_inc (struct sem * s) {
    //perform the V operation, increment the semaphore by 1, if the semaphore
    //value is now positive, any sleeping tasks are awakened.
    spin_lock(&s->lock);
    s->semaphore++;
    if (s->semaphore == 1) {

        while (s->prockBlockIndex != -1) {
            if(kill (pid_table[s->proc_block[s->prockBlockIndex]], SIGUSR1)<0) {
                fprintf(stderr,"ERROR: Failed to send signal to process %d: %s\n",pid_table[s-
>proc_block[s->prockBlockIndex]], strerror(errno));
                exit(EXIT_FAILURE);
            }
            s->prockBlockIndex--;
        }

    }
    spin_unlock(&s->lock);
}
```

sem.h:

```
#ifndef __SEM_H__
#define __SEM_H__

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

#include "spinlock.h"

#define N_PROC 64
```

```
extern int my_procnum;
extern pid_t * pid_table;

struct sem {
                                                            spinlock lock;
                                                            int semaphore;
                                                            int prockBlockIndex;
                                                            int proc_block[N_PROC];
                                                            sigset_t mask_block;
};

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);

#endif
```

Makefile:

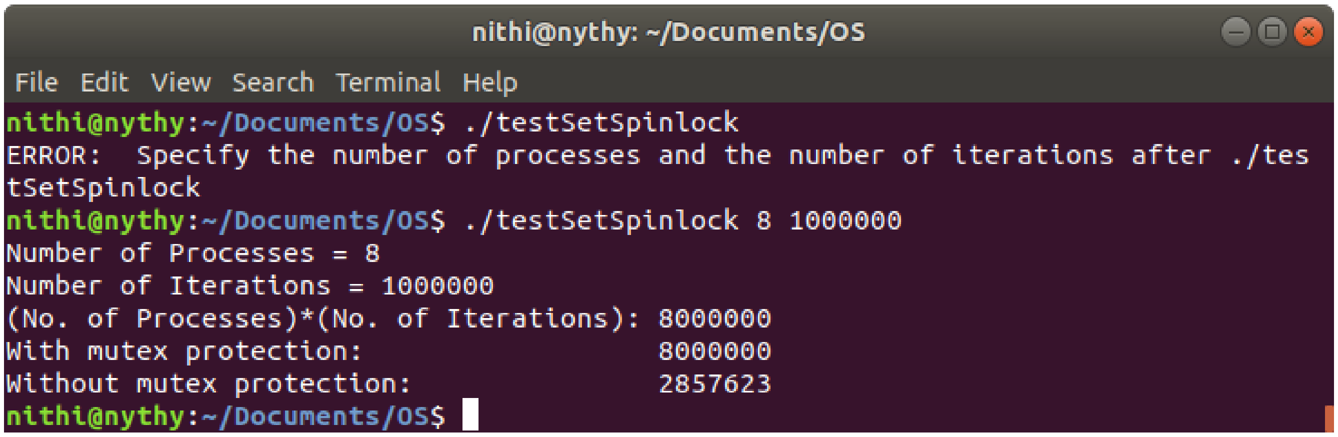
```
ALL: testSetSpinlock testFifo

testSetSpinlock: testSetSpinlock.c  spinlock.h spinlock.c tas64.s tas.h
	gcc -o testSetSpinlock spinlock.c testSetSpinlock.c tas64.s tas.h

testFifo:testFifo.c sem.c sem.h fifo.c fifo.h tas64.s spinlock.c
	gcc -o testFifo testFifo.c sem.c fifo.c  tas64.s spinlock.c
```

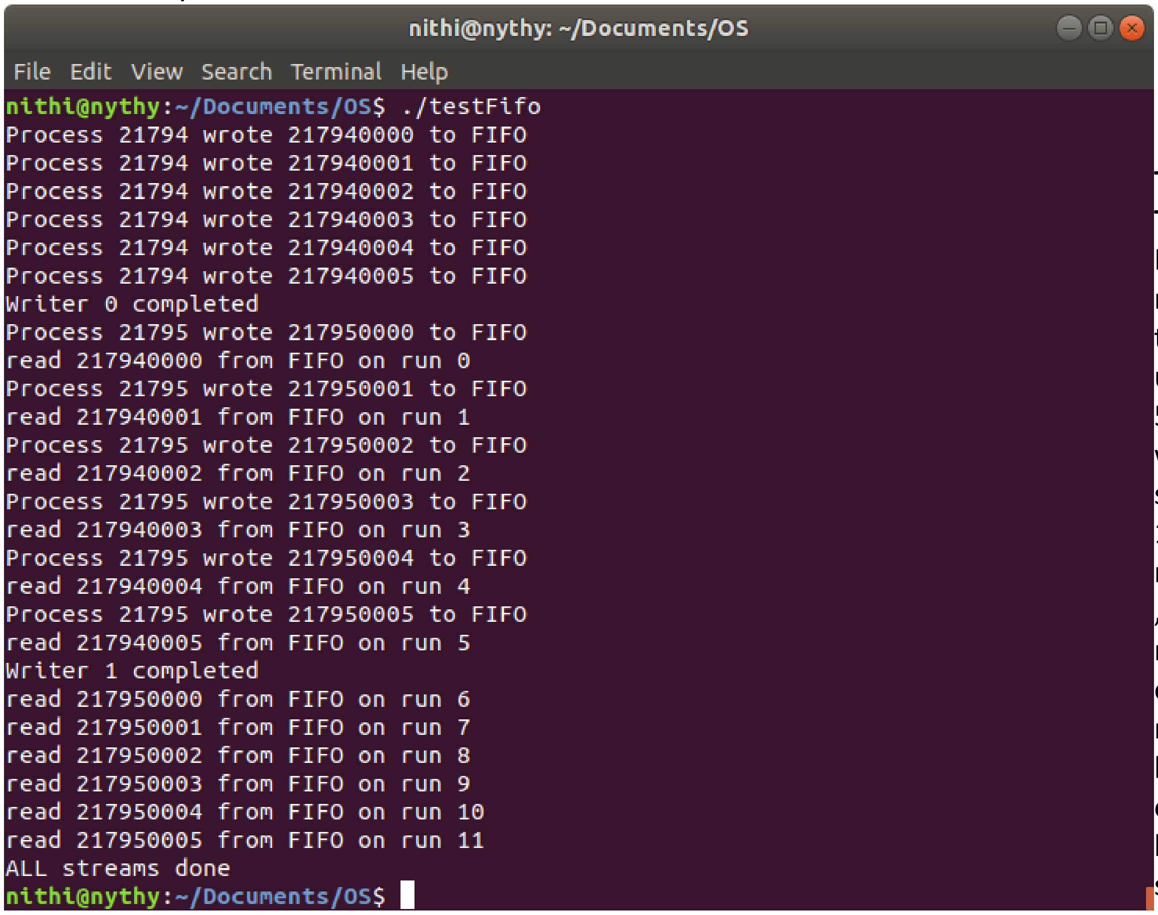
First Test:

simple test program that creates a shared memory region, spawns a bunch of processes sharing it, and does something non-atomic

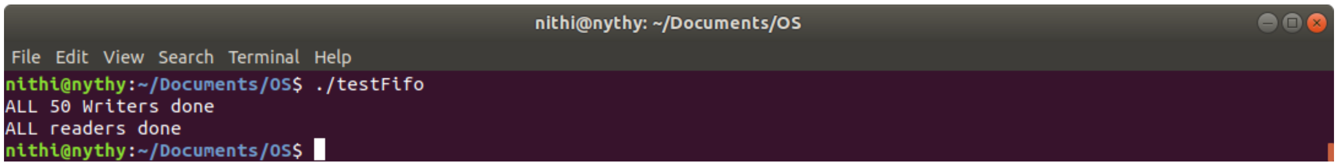

```
nithi@nythy:~/Documents/OS$ ./testSetSpinlock
ERROR:  Specify the number of processes and the number of iterations after ./tes
tSetSpinlock
nithi@nythy:~/Documents/OS$ ./testSetSpinlock 8 1000000
Number of Processes = 8
Number of Iterations = 1000000
(No. of Processes)*(No. of Iterations): 8000000
With mutex protection:                  8000000
Without mutex protection:               2857623
nithi@nythy:~/Documents/OS$
```

Second Test:

Shows that the implementation works for 2 writers and 1 reader.

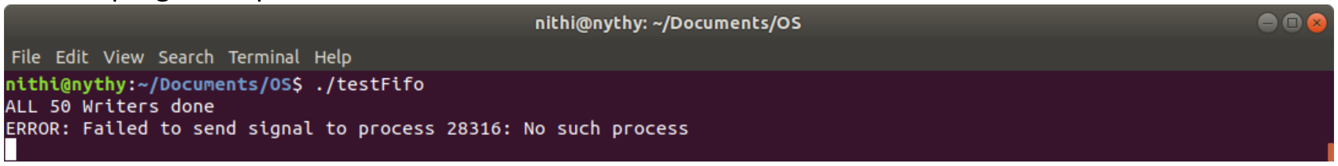

```
nithi@nythy:~/Documents/OS$ ./testFifo
Process 21794 wrote 217940000 to FIFO
Process 21794 wrote 217940001 to FIFO
Process 21794 wrote 217940002 to FIFO
Process 21794 wrote 217940003 to FIFO
Process 21794 wrote 217940004 to FIFO
Process 21794 wrote 217940005 to FIFO
Writer 0 completed
Process 21795 wrote 217950000 to FIFO
read 217940000 from FIFO on run 0
Process 21795 wrote 217950001 to FIFO
read 217940001 from FIFO on run 1
Process 21795 wrote 217950002 to FIFO
read 217940002 from FIFO on run 2
Process 21795 wrote 217950003 to FIFO
read 217940003 from FIFO on run 3
Process 21795 wrote 217950004 to FIFO
read 217940004 from FIFO on run 4
Process 21795 wrote 217950005 to FIFO
read 217940005 from FIFO on run 5
Writer 1 completed
read 217950000 from FIFO on run 6
read 217950001 from FIFO on run 7
read 217950002 from FIFO on run 8
read 217950003 from FIFO on run 9
read 217950004 from FIFO on run 10
read 217950005 from FIFO on run 11
ALL streams done
nithi@nythy:~/Documents/OS$
```

Third Test: Implementation using 50 writers and 1 reader, I removed the normal output lines so they didn't clutter the screen.


```
nithi@nythy:~/Documents/OS$ ./testFifo
ALL 50 Writers done
ALL readers done
nithi@nythy:~/Documents/OS$
```

Fourth Test:

I removed the spinlock from
void fifo_wr (struct fifo * f, unsigned long d);
and
unsigned long fifo_rd (struct fifo * f);
and the program reported and error as shown bellow:


nithi@nythy: ~/Documents/OS
File Edit View Search Terminal Help

```
nithi@nythy:~/Documents/OS$ ./testFifo
ALL 50 Writers done
ERROR: Failed to send signal to process 28316: No such process
```