ECE357, Computer Operating Systems – Problem Set #6

Nikola Janjušević

December 20, 2018

EXAMPLE BREAKAGE

The `acidtest` code (presented in the next section) spawns several child processes that write to a FIFO. Writing happens as a sequence of datums containing the child processes virtual process number `n_vp` and sequence number `k`. The parent process reads from the FIFO and prints out the contents of the datums as they are read. In normal operation of the `acidtest` program, datums are printed by the parent process in a seemingly random order, however, no datum from a single child process is ever out of order. `acidbreak.txt` shows a sample output of the program `acidtest.c` when the write mutex-lock is removed from the FIFO structure (removing lines 14 and 19 from `fifo.c`).

acidbreak.txt

```
$ ./acidtest.out
...
n_vp: 8,      k: 8388
n_vp: 12,     k: 8202
n_vp: 3,      k: 8363
n_vp: 4,      k: 8544
n_vp: 3,      k: 8364
n_vp: 6,      k: 8322
n_vp: 9,      k: 7966
n_vp: 5,      k: 8198
n_vp: 9,      k: 7967
n_vp: 6,      k: 8323
n_vp: 4,      k: 8545
n_vp: 12,     k: 8203
n_vp: 5,      k: 8200
n_vp: 8,      k: 8389
...
```

If we `grep` the output stream of the `acidtest` program for a single virtual process, we can see that the absence of the write mutex-lock causes the datums to be out of order. `acidbreak_grep.txt` shows a sample of the previous `acidtest` output when grepped for virtual process number 13.

acidbreak_grep.txt

```
$ ./acidtest.out | grep 'n_vp: 13'
...
n_vp: 13,     k: 9996
n_vp: 13,     k: 9997
n_vp: 13,     k: 9998
n_vp: 13,     k: 9999
n_vp: 13,     k: 9240
n_vp: 13,     k: 9241
n_vp: 13,     k: 9242
n_vp: 13,     k: 9243
n_vp: 13,     k: 9244
n_vp: 13,     k: 9245
...
```

ACID TEST

acidtest.c

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

#define N_ITER 1e4
#define N_PROC 16
#define MEM_SIZE 0x10000

int main(int argc, char **argv)
{
    int i,j,k, n_vp;
    unsigned long d=0;
    struct fifo *f;

    if( (f=mmap(NULL,MEM_SIZE,PROT_READ|PROT_WRITE,\
        MAP_SHARED|MAP_ANONYMOUS,-1,0))==MAP_FAILED )
    {
        perror("mmap error");
        exit(-1);
    }

    fifo_init(f);

    for(i=0;i<N_PROC;i++)
    {
        switch( fork() )
        {
            case -1:
                perror("fork error");
                exit(-1);
            case 0:
                for(j=0;j<N_ITER;j++)
                {
                    d = (unsigned long)i<<32 | (unsigned long)j;
                    fifo_wr(f,d);
                }
                exit(0);
            default:
                break;
        }
    }
    for(j=0;j<N_PROC*N_ITER;j++)
    {
        d = fifo_rd(f);
        n_vp = (int)(d>>32);
        k = (int)d;
        printf("n_vp: %d,\t k: %d\n",n_vp,k);
    }
    while(wait(NULL)>0);
    return 0;
}
```

FIFO

fifo.h

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

#define MYFIFO_BUFSIZ 0x1000
struct fifo
{
    struct sem sfifo_rd;
    struct sem sfifo_wr;
    struct sem smutex_rd;
    struct sem smutex_wr;
    unsigned long buf[MYFIFO_BUFSIZ];
    int writr;
    int rditr;
};
void fifo_init(struct fifo *f);
void fifo_wr(struct fifo *f, unsigned long d);
unsigned long fifo_rd(struct fifo *f);
#define __FIFO_H
#endif
```

fifo.c

```
#include "fifo.h"

void fifo_init(struct fifo *f)
{
    sem_init(&f->sfifo_wr,MYFIFO_BUFSIZ-1);
    sem_init(&f->sfifo_rd,0);
    sem_init(&f->smutex_wr,1);
    sem_init(&f->smutex_rd,1);
    f->writr=0; f->rditr=0;
}

void fifo_wr(struct fifo *f, unsigned long d)
{
    sem_wait(&f->smutex_wr);
        sem_wait(&f->sfifo_wr);
        f->buf[f->writr++] = d;
        f->writr %= MYFIFO_BUFSIZ;
        sem_inc(&f->sfifo_rd);
    sem_inc(&f->smutex_wr);
}

unsigned long fifo_rd(struct fifo *f)
{
    unsigned long ret;
    sem_wait(&f->smutex_rd);
        sem_wait(&f->sfifo_rd);
        ret = f->buf[f->rditr++];
        f->rditr %= MYFIFO_BUFSIZ;
        sem_inc(&f->sfifo_wr);
    sem_inc(&f->smutex_rd);
    return ret;
}
```

SEMAPHORE

sem.h

```
#ifndef __SEM_H
#include <sys/types.h>

#define MAX_WAIT 64
struct sem
{
    volatile int count;
    volatile pid_t waitstack[MAX_WAIT];
    volatile int waitlen;
    volatile char waitlock;
    volatile char countlock;
};
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);
#define __SEM_H
#endif
```

sem.c

```
#include "tas.h"
#include "spinlib.h"
#include "sem.h"
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

static void handler(int sn){}

static pid_t pid_pop(struct sem *s)
{
    if(s->waitlen<1)
        return -1;
    s->waitlen--;
    return s->waitstack[s->waitlen];
}

static pid_t pid_push(struct sem *s, pid_t pid)
{
    if(s->waitlen>=MAX_WAIT)
        return -1;
    s->waitstack[s->waitlen++] = pid;
    return 0;
}

static void block(struct sem *s)
{
    int i=0;
    sigset_t blk_mask, empty_mask;
    sigaddset(&blk_mask, SIGUSR1);
    sigemptyset(&empty_mask);

    spin_lock(&s->countlock);
        if(s->count>0)
        {
            spin_unlock(&s->countlock);
            return;
        }
    spin_lock(&s->waitlock);
        sigprocmask(SIG_BLOCK, &blk_mask, NULL);
        if( pid_push(s, getpid())<0 )
        {
            fprintf(stderr,"pid_push(s,%d) error, stack too large\n",getpid());
            exit(-1);
        }
    spin_unlock(&s->waitlock);
    spin_unlock(&s->countlock);

    sigsuspend(&empty_mask);
}

static void wake(struct sem *s)
{
    pid_t pid;
    spin_lock(&s->waitlock);
        while( (pid=pid_pop(s))>0 )
            kill(pid,SIGUSR1);
    spin_unlock(&s->waitlock);
}

void sem_init(struct sem *s, int count)
{
    s->count = count;
    s->countlock = 0;
    s->waitlock = 0;
    s->waitlen = 0;
    signal(SIGUSR1,handler);
}

int sem_try(struct sem *s)
{
    spin_lock(&s->countlock);
        if(s->count<1)
        {
            spin_unlock(&s->countlock);
            return 0;
        }
        s->count--;
    spin_unlock(&s->countlock);
    return 1;
}

void sem_wait(struct sem *s)
{
    while(!sem_try(s))
        block(s);
}

void sem_inc(struct sem *s)
{
    spin_lock(&s->countlock);
        s->count++;
        if(s->count==1)
            wake(s);
    spin_unlock(&s->countlock);
}
```

SPINLOCK

spinlib.h

```
#ifndef __SPINLIB_H
int spin_lock(volatile char *lock);
int spin_unlock(volatile char *lock);
#define __SPINLIB_H
#endif
```

spinlib.c

```
#include "spinlib.h"
#include "tas.h"
#include <sched.h>

int spin_lock(volatile char *lock)
{
    while(tas(lock))
        sched_yield();
    return 1;
}
int spin_unlock(volatile char *lock)
{
    *lock=0;
    return 0;
}
```

TAS

tas.h

```
#ifndef __TAS_H
int tas(volatile char *lock);
#define __TAS_H
#endif
```

tas.s

```
    .text
.global tas
    .type   tas,@function
tas:
    pushq   %rbp
    movq    %rsp, %rbp
    movq    $1, %rax
#APP
    lock;xchgb  %al,(%rdi)
#NO_APP
    movsbq  %al,%rax
    pop     %rbp
    ret
.Lfe1:
    .size   tas,.Lfe1-tas
```

MAKEFILE

Makefile

```
spintest.out:
    gcc -o spintest.out spintest.c spinlib.c tas.s
semtest.out:
    gcc -o semtest.out semtest.c sem.c spinlib.c tas.s
fifotest.out:
    gcc -o fifotest.out fifotest.c fifo.c sem.c spinlib.c tas.s
acidtest.out:
    gcc -o acidtest.out acidtest.c fifo.c sem.c spinlib.c tas.s
```