**Spinlock.h:**

#include <stdio.h>

#include <unistd.h>

#include <string.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <sys/mman.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <fcntl.h>

#include <errno.h>

#include <signal.h>

typedef struct spinlock{

   volatile char primitive\_lock;

}spinlock;

int tas(volatile char \*lock);

void spin\_lock(struct spinlock \*l);

void spin\_unlock(struct spinlock \*l);

**Spinlock.c:**

#include "spinlock.h"

void spin\_lock(struct spinlock \*l){

while(tas(&(l->primitive\_lock)) != 0){

  ;

}

}

void spin\_unlock(struct spinlock \*l){

l->primitive\_lock=0;

}

**Spintest.c (Problem 2):**

#include "spinlock.h"

int i;

int k;

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

 if(argc != 3){

  fprintf(stderr, "CRITICAL ERROR: Incorrect number of arguments (3 required)\n");

  exit(1);

 }

 long long unsigned int procnum = atoll(argv[1]);

 long long unsigned int iternum = atoll(argv[2]);

 printf("procnum = %llu\niternum = %llu\n", fknum, iternum);

 int \*map = mmap(NULL, 4096, PROT\_READ | PROT\_WRITE, MAP\_ANONYMOUS | MAP\_SHARED, 0, 0 );

 if(map< 0){

  fprintf(stderr, "CRITICAL ERROR: Unable to MMAP ANON page: %s\n", strerror(errno));

  exit(1);

 }

 map[0] = 0;

 spinlock \*lock;

 lock = (spinlock \*)(map+sizeof(spinlock));

 lock->primitive\_lock= map[1];

 pid\_t pid[procnum];

 for (i = 0; i < procnum; i++){

//fork

if((pid[i] = fork()) < 0){

  //fork ERROR

    fprintf(stderr, "NON-CRITICAL ERROR: Unable to fork id#%d: %s\n",i, strerror(errno));

    return(1);

}

if(pid[i] == 0){

    spin\_lock(lock);

  for(k = 0; k < iternum; k++){

    map[0]++;

  }

  spin\_unlock(lock);

  exit(0);

}

 }

 for(i = 0; i < fknum; i++){

wait(0);

 }

 printf("%d\n", map[0]);

 return(0);

}

**Cv.h:**

#include "spinlock.h"

#define CV\_MAXPROC 64

static int wait\_count;

static int swakeup\_count;

static int wakeup\_count;

typedef struct cv {

   int count;

   spinlock lock;

   pid\_t pid[CV\_MAXPROC];

   sigset\_t sigmask;

} cv;

void cv\_init(struct cv \*cv);

/\* Initialize any internal data structures in cv so that it is ready for

\* use.  The initial condition is that nobody is waiting for this cv.

\* You can probably arrange your struct cv so that all-0 bytes is

\* the initialization condition.

\*/

void cv\_wait(struct cv \*cv, struct spinlock \*mutex);

/\* This will be called with the spinlock mutex held by the caller (otherwise

\* results will be undefined).  Atomically record within the internals

\* of cv that the caller is going to sleep, release the mutex, and

\* go to sleep (see text below).  After waking up, re-acquire the mutex

\* before returning to the caller

\*/

int cv\_broadcast(struct cv \*cv);

/\* Wake up any and all waiters (sleepers) on this cv.  If there are no waiters

\* the call has no effect and is not "remembered" for the next time that

\* someone calls cv\_wait.  cv\_broadcast should be called with the same mutex

\* held that protects cv\_wait, as discussed in lecture notes under "Lost

\* Wakup", but note that cv\_broadcast does not take a mutex as a parameter.

\* Return value: the number of sleepers that were awoken.

\*/

int cv\_signal(struct cv \*cv);

/\* Exactly the same as cv\_broadcast except at most one sleeper is awoken.

\* Your choice how to pick which one if more than one candidate

\*/

**Cv.c:**

#include "cv.h"

void sighandler(int signo){;}

void cv\_init(struct cv \*cv){

 int \*map = mmap(NULL, 4096, PROT\_READ | PROT\_WRITE, MAP\_ANONYMOUS| MAP\_SHARED, 0, 0);

 int i;

 if(map< 0){

  fprintf(stderr, "CRITICAL ERROR: Unable to MMAP ANONYMOUS page: %s\n", strerror(errno));

  exit(1);

 }

 spinlock \*lock;

 lock = (spinlock \*)(map+sizeof(spinlock));

 cv->lock=\*lock;

for(i = 0; i < CV\_MAXPROC; i++){

  cv->pid[i] = 0;

}

cv->count = 0;

signal(SIGUSR1, sighandler);

sigfillset(&cv->sigmask);

sigdelset(&cv->sigmask, SIGUSR1);

//SIGNAL STUFF

}

void cv\_wait(struct cv \*cv, struct spinlock \*mutex){

 if(cv->count >= CV\_MAXPROC){

  fprintf(stderr, "CRITICAL ERROR: Too many processes running\n");

  exit(1);

 }

 spin\_lock(&cv->lock);

 cv->pid[cv->count] = getpid();

 cv->count++;

 spin\_unlock(&cv->lock);

 spin\_unlock(mutex);

 //SIGNALS

 sigprocmask(SIG\_BLOCK, &cv->sigmask, NULL);

 sigsuspend(&cv->sigmask);

 if(cv->count > 0){

spin\_lock(&cv->lock);

cv->pid[cv->count - 1] = 0;

cv->count--;

spin\_unlock(&cv->lock);

spin\_lock(mutex);

return;

 }

 sigprocmask(SIG\_UNBLOCK, &cv->sigmask, NULL);

 spin\_lock(mutex);

}

int cv\_broadcast(struct cv \*cv){

 int wkcount = 0;

 int i;

 spin\_lock(&cv->lock);

 if(cv->count == 0){

spin\_unlock(&cv->lock);

return(0);

 }

 for(i = 0; i < CV\_MAXPROC; i++){

if(cv->pid[i] > 0){

  kill(cv->pid[i], SIGUSR1);

  wkcount++;

}

 }

 spin\_unlock(&cv->lock);

 return(wkcount);

}

int cv\_signal(struct cv \*cv){

 int wkcount = 0;

 spin\_lock(&cv->lock);

 if(cv->count == 0){

spin\_unlock(&cv->lock);

return(0);

 }

 kill(cv->pid[cv->count - 1], SIGUSR1);

 wkcount++;

 spin\_unlock(&cv->lock);

 return(wkcount);

}

**Cvtest.c:**

#include "cv.h"

int pid, i;

int main(){

 spinlock testlock;

 cv \*cv;

 cv = (struct cv \*)mmap(NULL, sizeof(cv), PROT\_READ | PROT\_WRITE, MAP\_SHARED | MAP\_ANONYMOUS, -1, 0);

 cv\_init(cv);

 for(i = 0; i < 3; i++){

if((pid = fork()) < 0){

  //fork ERROR

    fprintf(stderr, "NON-CRITICAL ERROR: Unable to fork id#%d: %s\n",i, strerror(errno));

    return(1);

}

if(pid == 0){

  printf("Fork Succeeded, Wait\n");

  cv\_wait(cv, &testlock);

  exit(0);

}

 }

 sleep(2);

 printf("Waking up everything\n");

 int wakeup = cv\_broadcast(cv);

 for(i = 0; i < 2; i++){

printf("Waiting for children to exit\n");

wait(0);

 }

 printf("Wakeup Num = %d\n", wakeup);

 return(0);

}

**Fifo.h:**

#include "cv.h"

#define MYFIFO\_BUFSIZ 1024

typedef struct fifo{

unsigned long buf[MYFIFO\_BUFSIZ];

int state;

int nextread;

int nextwrite;

spinlock fifolock;

cv wr;

cv rd;

}fifo;

void fifo\_init(struct fifo \*f);

/\* Initialize the shared memory FIFO \*f including any required underlying

\* initializations (such as calling cv\_init).  The FIFO will have a static

\* fifo length of MYFIFO\_BUFSIZ elements.   #define this in fifo.h.

\* A value of 1K is reasonable.

\*/

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

/\* Enqueue the data word d into the FIFO, blocking unless and until the

\* FIFO has room to accept it. (i.e. block until !full)

\*/

unsigned long fifo\_rd(struct fifo \*f);

/\* Dequeue the next data word from the FIFO and return it.  Block unless

\* and until there are available words.  (i.e. block until !empty)

\*/

**Fifo.c:**

#include "fifo.h"

int i, z;

void fifo\_init(struct fifo \*fifo){

 cv \*rdmap = NULL;

 cv \*wrmap = NULL;

 rdmap = (cv \*)mmap(NULL, sizeof(cv), PROT\_READ | PROT\_WRITE, MAP\_SHARED | MAP\_ANONYMOUS, -1, 0);

 if(rdmap < 0){

fprintf(stderr, "CRITICAL ERROR: Unable to mmap ANONYMOUS file for read: %s\n", strerror(errno));

exit(1);

 }

 wrmap = (cv \*)mmap(NULL, sizeof(cv), PROT\_READ | PROT\_WRITE, MAP\_SHARED | MAP\_ANONYMOUS, -1, 0);

 if(wrmap < 0){

fprintf(stderr, "CRITICAL ERROR: Unable to mmap ANONYMOUS file for read: %s\n", strerror(errno));

exit(1);

 }

 fifo->rd = \*rdmap;

 cv\_init(&fifo->rd);

 fifo->nextread = 0;

 fifo->wr = \*wrmap;

 cv\_init(&fifo->wr);

 fifo->nextwrite = 0;

 fifo->state = 0;

 fifo->fifolock.primitive\_lock = 0;

}

void fifo\_wr(struct fifo \*fifo, unsigned long d){

 spin\_lock(&fifo->fifolock);

 while(fifo->state >= MYFIFO\_BUFSIZ){

cv\_wait(&fifo->wr, &fifo->fifolock);

 }

 fifo->buf[fifo->nextwrite++] = d;

 fifo->nextwrite %= MYFIFO\_BUFSIZ;

 fifo->state++;

 //SIGNALING

 cv\_signal(&fifo->rd);

 //UNLOCKING

 spin\_unlock(&fifo->fifolock);

}

unsigned long fifo\_rd(struct fifo \*fifo){

 unsigned long fiforead;

 spin\_lock(&fifo->fifolock);

 while(fifo->state <= 0){

printf("Reader Stream %d complete\n", ++z);

cv\_wait(&fifo->rd, &fifo->fifolock);

 }

 fiforead = fifo->buf[fifo->nextread++];

 fifo->nextread %= MYFIFO\_BUFSIZ;

 fifo->state--;

 //SIGNALING

 cv\_signal(&fifo->wr);

 //UNLOCKING

 spin\_unlock(&fifo->fifolock);

 return(fiforead);

}

**Fifotest.c:**

#include "fifo.h"

int procno, i, k, reader, n, w;

int numwriters = 5;

int writelen = MYFIFO\_BUFSIZ;

int main(){

 fifo \*fif;

fif = (fifo \* )mmap (NULL, sizeof (fifo), PROT\_READ | PROT\_WRITE, MAP\_SHARED | MAP\_ANONYMOUS, -1, 0);

 if(fif < 0){

fprintf(stderr, "CRITICAL ERROR: Unable to mmap ANONYMOUS page for fifo: %s\n", strerror(errno));

exit(1);

 }

 fifo\_init(fif);

 printf("Beginning test with %d writers, %d items each\n", numwriters, writelen);

 pid\_t pid[numwriters];

 for(i = 0; i < numwriters; i++){

if((pid[i] = fork()) < 0){

  //FORK ERROR

    fprintf(stderr, "NON-CRITICAL ERROR: Unable to fork id#%d: %s\n",i, strerror(errno));

    return(1);

}

if(pid[i] == 0){

  //CHILD

  procno = i;

  unsigned long writebuf[writelen];

  for(k = 0; k < writelen; k++){

    writebuf[k] = getpid()\*10000 + k;

    fifo\_wr(fif, writebuf[k]);

  }

  printf("Writer %d completed\n", i);

  exit(0);

}

 }

 if((reader = fork()) < 0){

//FORK ERROR

    fprintf(stderr, "NON-CRITICAL ERROR: Unable to fork: %s\n", strerror(errno));

    return(1);

 }

 if(reader == 0){

//CHILD

procno = numwriters;

unsigned long readbuf[numwriters\*writelen];

int nread = numwriters\*writelen;

for(w = 0; w < nread; w++){

  readbuf[w] = fifo\_rd(fif);

}

printf("All streams done\n");

 }

 for(n = 0; n < numwriters + 1; n++){

printf("Waiting for children to die\n");

wait(0);

 }

 return(0);

}

**Sample Outputs:**

Spinlock Test(Problems 1 & 2):

**WITH LOCK**

ali@ali-VirtualBox:~/Documents/PS6$ ./a.out 4 100000

procnum = 8

iternum = 100000

800000

**WITHOUT LOCK**

ali@ali-VirtualBox:~/Documents/PS6$ ./a.out 8 100000

procnum = 8

iternum = 100000

500000

Conditional Variable Test (Problem 3):

ali@ali-VirtualBox:~/Documents/PS6$ ./a.out

Fork Succeeded, Wait

Fork Succeeded, Wait

Fork Succeeded, Wait

Waking up everything

Waiting for children to exit

Waiting for children to exit

Wakeup Num = 3

FIFO TEST (Problems 4 & 5):

ali@ali-VirtualBox:~/Documents/PS6$ ./a.out

Beginning test with 5 writers, 1024 items each

Waiting for children to die

Writer 4 completed

Reader Stream 1 complete

Waiting for children to die

Writer 0 completed

Reader Stream 2 complete

Waiting for children to die

Writer 1 completed

Reader Stream 3 complete

Waiting for children to die

Writer 2 completed

Waiting for children to die

Reader Stream 4 complete

Writer 3 completed

Waiting for children to die

All streams done

Waiting for children to die

Waiting for children to die

Waiting for children to die

Waiting for children to die

Waiting for children to die

Waiting for children to die

Short FIFO Test (10 Writers):

ali@ali-VirtualBox:~/Documents/PS6$ ./a.out

Beginning test with 10 writers, 1024 items each

Writer 4 completed

Reader Stream 1 complete

Waiting for children to die

Writer 7 completed

Reader Stream 2 complete

Writer 8 completed

Reader Stream 3 complete

Writer 9 completed

Reader Stream 4 complete

Writer 6 completed

Reader Stream 5 complete

Writer 1 completed

Reader Stream 6 complete

Writer 2 completed

Reader Stream 7 complete

Writer 5 completed

Reader Stream 8 complete

Writer 0 completed

Reader Stream 9 complete

Writer 3 completed

All streams done

Waiting for children to die