**Operating Systems HW5**

**1st Q**: The first problem is just to implement and test a solution to the fork/join   
problem, as described in the text. Even though this solution is described in the   
text, the act of typing it in on your own is worthwhile; even Bach would rewrite   
Vivaldi, allowing one soon-to-be master to learn from an existing one. See fork-  
join.c for details. Add the call sleep (1) to the child to ensure it is working.

**Answer:**

**Code:**

#include <stdio.h>

#include <unistd.h>

#include <pthread.h>

#include "common\_threads.h"

sem\_t s;

void \*child(void \*arg) {

printf("child\n");

// use semaphore here sleep(1);

Sem\_post(&s);

return NULL;

}

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

pthread\_t p;

printf("parent: begin\n");

// init semaphore here

Sem\_init(&s, 0);

Pthread\_create(&p, NULL, child, NULL);

// use semaphore here

Sem\_wait(&s);

printf("parent: end\n");

return 0;

}

**Output:**

Text

Description automatically generated

**2nd Q:** Let’s now generalize this a bit by investigating the rendezvous problem. The   
problem is as follows: you have two threads, each of which are about to   
enter the rendezvous point in the code. Neither should exit this part of the   
code before the other enters it. Consider using two semaphores for this task,   
and see rendezvous.c for detail.s

**Answer:**

**Code:**

#include <stdio.h>

#include <unistd.h>

#include "common\_threads.h"

// If done correctly, each child should print their "before" message

// before either prints their "after" message. Test by adding sleep(1)

// calls in various locations.

sem\_t s1, s2;

void \*child\_1(void \*arg) {

printf("child 1: before\n");

// what goes here?

Sem\_post(&s2);

Sem\_wait(&s1);

sleep(1);

printf("child 1: after\n");

return NULL;

}

void \*child\_2(void \*arg) {

printf("child 2: before\n");

// what goes here?

Sem\_post(&s1);

Sem\_wait(&s2);

printf("child 2: after\n");

return NULL;

}

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

pthread\_t p1, p2;

printf("parent: begin\n");

// init semaphores here

Sem\_init(&s1, 0);

Sem\_init(&s2, 0);

Pthread\_create(&p1, NULL, child\_1, NULL);

Pthread\_create(&p2, NULL, child\_2, NULL);

Pthread\_join(p1, NULL);

Pthread\_join(p2, NULL);

printf("parent: end\n");

return 0;

}

**Output:**

Text

Description automatically generated

**3rd Q:** Now go one step further by implementing a general solution to barrier syn-   
chronization. Assume there are two points in a sequential piece of code,   
called P1 and P2. Putting a barrier between P1 and P2 guarantees that all   
threads will execute P1 before any one thread executes P2. Your task: write   
the code to implement a barrier()   
function that can be used in this man- ner.   
It is safe to assume you know N (the total number of threads in the running   
program) and that all N threads will try to enter the barrier. Again, you should   
likely use two semaphores to achieve the solution, and some other integers   
to count things. See barrier.c for details.

**Answer:**

**Code:**

#include <assert.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include "common\_threads.h"

// If done correctly, each child should print their "before" message

// before either prints their "after" message. Test by adding sleep(1)

// calls in various locations.

// You likely need two semaphores to do this correctly, and some

// other integers to track things.

typedef struct \_\_barrier\_t {

// add semaphores and other information here

int total\_threads;

int threads\_arrived;

sem\_t sem\_lock;

sem\_t sem\_barr;

} barrier\_t;

// the single barrier we are using for this program

barrier\_t b;

void barrier\_init(barrier\_t \*b, int num\_threads) {

// initialization code goes here

b->total\_threads = num\_threads;

sem\_init(&b->sem\_lock,0,1);

sem\_init(&b->sem\_barr,0,0);

b->threads\_arrived = 0;

}

void barrier(barrier\_t \*b) {

// barrier code goes here

sem\_wait(&b->sem\_lock);

if(b->threads\_arrived < b->total\_threads-1)

{

b->threads\_arrived++;

sleep(1);

sem\_post(&b->sem\_lock);

sem\_wait(&b->sem\_barr);

}

sem\_post(&b->sem\_barr);

}

// XXX: don't change below here (just run it!)

typedef struct \_\_tinfo\_t {

int thread\_id;

} tinfo\_t;

void \*child(void \*arg) {

tinfo\_t \*t = (tinfo\_t \*) arg;

printf("child %d: before\n", t->thread\_id);

barrier(&b);

printf("child %d: after\n", t->thread\_id);

return NULL;

}

// run with a single argument indicating the number of

// threads you wish to create (1 or more)

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

assert(argc == 2);

int num\_threads = atoi(argv[1]);

assert(num\_threads > 0);

pthread\_t p[num\_threads];

tinfo\_t t[num\_threads];

printf("parent: begin\n");

barrier\_init(&b, num\_threads);

int i;

for (i = 0; i < num\_threads; i++) {

t[i].thread\_id = i;

Pthread\_create(&p[i], NULL, child, &t[i]);

}

for (i = 0; i < num\_threads; i++)

Pthread\_join(p[i], NULL);

printf("parent: end\n");

return 0;

}

**Output:**

Text

Description automatically generated

**4th Q:** Now let’s solve the reader-writer problem, also as described in the text. In   
this first take, don’t worry about starvation. See the code in reader-writer.c for   
details. Add sleep() calls to your code to demonstrate it works as you expect.   
Can you show the existence of the starvation problem?

**Answer:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include "common\_threads.h"

// Your code goes in the structure and functions below

typedef struct \_\_rwlock\_t {

sem\_t lock;

sem\_t writelock;

int readers;

} rwlock\_t;

void rwlock\_init(rwlock\_t \*rw) {

rw->readers = 0;

sem\_init(&rw->lock, 0, 1);

sem\_init(&rw->writelock, 0, 1);

}

void rwlock\_acquire\_readlock(rwlock\_t \*rw) {

sem\_wait(&rw->lock);

rw->readers++;

if(rw->readers == 1)

sem\_wait(&rw->writelock);

sem\_post(&rw->lock);

}

void rwlock\_release\_readlock(rwlock\_t \*rw) {

sem\_wait(&rw->lock);

rw->readers--;

if(rw->readers == 0)

sem\_post(&rw->writelock);

sem\_post(&rw->lock);

}

void rwlock\_acquire\_writelock(rwlock\_t \*rw) {

sem\_wait(&rw->writelock);

}

void rwlock\_release\_writelock(rwlock\_t \*rw) {

sem\_post(&rw->writelock);

}

// Don't change the code below (just use it!)

int loops;

int value = 0;

rwlock\_t lock;

void \*reader(void \*arg) {

int i;

for (i = 0; i < loops; i++) {

rwlock\_acquire\_readlock(&lock);

printf("read %d\n", value);

sleep(2);

rwlock\_release\_readlock(&lock);

}

return NULL;

}

void \*writer(void \*arg) {

int i;

for (i = 0; i < loops; i++) {

rwlock\_acquire\_writelock(&lock);

value++;

printf("write %d\n", value);

rwlock\_release\_writelock(&lock);

}

return NULL;

}

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

assert(argc == 4);

int num\_readers = atoi(argv[1]);

int num\_writers = atoi(argv[2]);

loops = atoi(argv[3]);

pthread\_t pr[num\_readers], pw[num\_writers];

rwlock\_init(&lock);

printf("begin\n");

int i;

for (i = 0; i < num\_readers; i++)

Pthread\_create(&pr[i], NULL, reader, NULL);

for (i = 0; i < num\_writers; i++)

Pthread\_create(&pw[i], NULL, writer, NULL);

for (i = 0; i < num\_readers; i++)

Pthread\_join(pr[i], NULL);

for (i = 0; i < num\_writers; i++)

Pthread\_join(pw[i], NULL);

printf("end: value %d\n", value);

return 0;

}

**Output:**

Text

Description automatically generated

As we can see, the writer is waiting for all the readers to complete. Therefore, the writer starvation occurred here.

**5th Q:** Let’s look at the reader-writer problem again, but this time, worry about   
starvation. How can you ensure that all readers and writers eventually make   
progress? See reader-writer-nostarve.c for details.

**Answer:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include "common\_threads.h"

// Your code goes in the structure and functions below

typedef struct \_\_rwlock\_t {

sem\_t lock;

sem\_t writelock;

int readers;

} rwlock\_t;

void rwlock\_init(rwlock\_t \*rw) {

rw->readers = 0;

sem\_init(&rw->lock, 0, 1);

sem\_init(&rw->writelock, 0, 1);

}

void rwlock\_acquire\_readlock(rwlock\_t \*rw) {

sem\_wait(&rw->lock);

rw->readers++;

if(rw->readers == 1)

sem\_wait(&rw->writelock);

sem\_post(&rw->lock);

}

void rwlock\_release\_readlock(rwlock\_t \*rw) {

sem\_wait(&rw->lock);

rw->readers--;

if(rw->readers == 0)

sem\_post(&rw->writelock);

sem\_post(&rw->writelock);

sem\_post(&rw->lock);

}

void rwlock\_acquire\_writelock(rwlock\_t \*rw) {

sem\_wait(&rw->writelock);

}

void rwlock\_release\_writelock(rwlock\_t \*rw) {

sem\_post(&rw->writelock);

}

// Don't change the code below (just use it!)

int loops;

int value = 0;

rwlock\_t lock;

void \*reader(void \*arg) {

int i;

for (i = 0; i < loops; i++) {

rwlock\_acquire\_readlock(&lock);

printf("read %d\n", value);

rwlock\_release\_readlock(&lock);

}

return NULL;

}

void \*writer(void \*arg) {

int i;

for (i = 0; i < loops; i++) {

rwlock\_acquire\_writelock(&lock);

value++;

printf("write %d\n", value);

rwlock\_release\_writelock(&lock);

}

return NULL;

}

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

assert(argc == 4);

int num\_readers = atoi(argv[1]);

int num\_writers = atoi(argv[2]);

loops = atoi(argv[3]);

pthread\_t pr[num\_readers], pw[num\_writers];

rwlock\_init(&lock);

printf("begin\n");

int i;

for (i = 0; i < num\_readers; i++)

Pthread\_create(&pr[i], NULL, reader, NULL);

for (i = 0; i < num\_writers; i++)

Pthread\_create(&pw[i], NULL, writer, NULL);

for (i = 0; i < num\_readers; i++)

Pthread\_join(pr[i], NULL);

for (i = 0; i < num\_writers; i++)

Pthread\_join(pw[i], NULL);

printf("end: value %d\n", value);

return 0;

}

**Output:**

Text

Description automatically generated

**6th Q:** Use semaphores to build a no-starve mutex, in which any thread that tries to   
acquire the mutex will eventually obtain it. See the code in mutex-nostarve.c   
for more information.

**Answer:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include "common\_threads.h"

// Here, you have to write (almost) ALL the code. Oh no!

// How can you show that a thread does not starve

// when attempting to acquire this mutex you build?

typedef struct \_\_ns\_mutex\_t {

sem\_t lock;

} ns\_mutex\_t;

void ns\_mutex\_init(ns\_mutex\_t \*m) {

sem\_init(&m->lock, 0, 1);

}

void ns\_mutex\_acquire(ns\_mutex\_t \*m) {

sem\_wait(&m->lock);

}

void ns\_mutex\_release(ns\_mutex\_t \*m) {

sem\_post(&m->lock);

}

ns\_mutex\_t n;

int c=0;

void \*worker(void \*arg) {

ns\_mutex\_acquire(&n);

c++;

ns\_mutex\_release(&n);

return NULL;

}

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

assert(argc==2);

int thread\_number = atoi(argv[1]);

pthread\_t p[thread\_number];

ns\_mutex\_init(&n);

printf("parent: begin\n");

int i;

for(i=0;i<thread\_number;i++)

{

pthread\_create(&p[i],NULL,worker,NULL);

}

int j;

for(j=0;j<thread\_number;j++)

{

pthread\_join(p[j],NULL);

}

printf("Counter value : %d \n",c);

printf("parent: end\n");

return 0;

}

**Output:**

Text

Description automatically generated