Homework Wan Huzaifah bin Wan Azhar

Answer:



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

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

**sem\_wait(&s1);**

**sem\_post(&s2);**

**sleep(1);**

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

return NULL;

}

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

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

**sem\_post(&s1);**

**sem\_wait(&s2);**

**sleep(1);**

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

return NULL;

}

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

pthread\_t p1, p2;

printf("parent: begin\n");

**sem\_init(&s1, 0, 0);**

**sem\_init(&s2, 0, 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;

}

Should print correctly before and then begin.



typedef struct \_\_barrier\_t {

sem\_t barrierlock;

sem\_t counterlock;

int NUM\_THREAD;

int counter;

} barrier\_t;

// the single barrier we are using for this program

barrier\_t b;

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

b->NUM\_THREAD = num\_threads;

b->counter = 0;

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

sem\_init(&b->counterlock, 0, 1); //Correctly increment counting, binary lock

}

void barrier(barrier\_t \*b) {

sem\_wait(&b->counterlock);

b->counter++;

sem\_post(&b->counterlock);

if (b->counter == b->NUM\_THREAD) {

sem\_post(&b->barrierlock); //Release one lock if all thread is finished

}

else {

sem\_wait(&b->barrierlock);

}

sem\_post(&b->barrierlock); //Release rest of lock

}

Running “./barrier 10” gives output:

parent: begin

child 0: before

child 1: before

child 4: before

child 2: before

child 3: before

child 5: before

child 9: before

child 8: before

child 7: before

child 6: before

child 6: after

child 0: after

child 5: after

child 2: after

child 1: after

child 3: after

child 4: after

child 9: after

child 8: after

child 7: after

parent: end



* Code is just like in Textbook.
* Running “./reader-writer 30 5 4” can starve writer as writer cannot start until there is no more reader.



typedef struct \_\_rwlock\_t {

sem\_t writelock;

sem\_t lock;

sem\_t readlock;

int readers;

} rwlock\_t;

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

rw->readers = 0;

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

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

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

}

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

sem\_wait(&rw->readlock);

sem\_post(&rw->readlock);

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->readlock);

sem\_wait(&rw->writelock);

}

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

sem\_post(&rw->writelock);

sem\_post(&rw->readlock);

}