Assignment 7

Adv C Programming Comprehensive Assignment

Title: Mastering Advanced C Programing Concepts - Semaphores

Objective: The objective of this assignment is to reinforce your understanding of advanced C programming concepts.

Assignment Task:

Implementing Semaphores

1. A producer-consumer example was introduced in the lecture. In the problem, a producer thread and a consumer thread shared a bounded buffer with n slots.

- The producer creates items and adds them to the buffer
- The consumer removes items from the buffer and consumes (uses) them
- If the buffer is full, the producer needs to wait
- If the buffer is empty, the consumer needs to wait

The example is slightly modified in the number of inner and outer loops as shown below (i.e. green lines) to examine the behavior of semaphores; however, the printed results seem incorrect. For instance, when you compile and execute the program, you can find these printed lines in the execution as follows. **Correct the printing code to monitor the behavior of consumer and producer in the program and discuss whether the semaphores correctly work or not.**

```
% ./assignment7_1
consumer: sum: 0, size: 0
consumer: sum: 145, size: 10
consumer: sum: 245, size: 13
producer: sum: 4950, size: 70
producer: sum: 4950, size: 170

% ./assignment7_1
consumer: sum: 45, size: 0
consumer: sum: 145, size: 11
consumer: sum: 245, size: 12
producer: sum: 4950, size: 70
producer: sum: 4950, size: 170
```

```
/*
* File: assignment7_1.c
* Owner: Yoonseok Yang
* Date: 04.25.2024
* Description: Implement the producer-consumer problem using semaphores
in C
*/

#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
typedef struct {
   int *buf;
   int capacity, head, tail;
   sem_t mutex;
   sem_t slots;
   sem_t items;
} sbuf_t;

void sbuf_init(sbuf_t* sp, int n) {
   sp->buf = (int*) calloc(n, sizeof(int));
   sp->capacity = n;
   sp->head = sp->tail = 0;
   sem_init(&sp->mutex, 0, 1);
   sem_init(&sp->slots, 0, n);
   sem_init(&sp->items, 0, 0);
}
```

```
void sbuf_deinit(sbuf_t *sp) {
    free(sp->buf);
    sem_destroy(&sp->mutex);
    sem_destroy(&sp->slots); sem_destroy(&sp->items);
}

int sbuf_size(sbuf_t *sp) {
    sem_wait(&sp->mutex); // access buffer after acquiring the lock
    int n = (sp->head + sp->capacity - sp->tail) % sp->capacity;
    sem_post(&sp->mutex);
    return n;
}

void sbuf_insert(sbuf_t *sp, int item) {
    sem_wait(&sp->slots); // wait while the buffer is full
    sem_wait(&sp->mutex); // access buffer after acquiring the lock
    sp->head = (sp->head + 1) % sp->capacity;
    sp->buf[sp->head] = item;
    sem_post(&sp->mutex);
    sem_post(&sp->items); // wake up consumer if it is suspended
}

int sbuf_remove(sbuf_t *sp) {
    sem_wait(&sp->items); // wait while the buffer is empty
    sem_wait(&sp->mutex); // access buffer after acquiring the lock
    sp->tail= (sp->tail + 1) % sp->capacity;
    int item = sp->buf[sp->tail];
    sem_post(&sp->mutex);
    sem_post(&sp->slots); // wake up producer if it is suspended
    return item;
}

void* producer(void* vargp) {
    sbuf_t *sp = (sbuf_t*)vargp;
    int i, j;
    for(i = 0; i < 2; i++) {
        long s = 0;
        for(j = 0; j < 100; j++)
            s += j, sbuf_insert(sp, j);
        printf("producer: sum: %ld, size: %d\n", s, sbuf_size(sp));
    }
    pthread_exit(NULL);
}

void* consumer(void* vargp) {
    sbuf_t *sp = (sbuf_t*)vargp;
    int i, j;
    for(i = 0; i < 3; i++) {
        long s = 0;
        for(j = 0; j < 10; j++)
            s += sbuf_remove(sp);
        printf("consumer: sum: %ld, size: %d\n", s, sbuf_size(sp));
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t tid_p, tid_c;
    sbuf_t sb;
    sbuf_init(&sb, 15000);
    pthread_create(&tid_p, NULL, producer, &sb);
    pthread_create(&tid_c, NULL, consumer, &sb);

    pthread_join(tid_p, NULL);
    pthread_join(tid_c, NULL);

    sbuf_deinit(&sb);
    return 0;
}
```

2. Another example of semaphores is a readers-writers problem, which is similar to the consumer-producer one.
The readers-writers problem includes a collection of concurrent threads that access a shared data. The reader threads only read the data and write threads only modify the data.

There are two approaches to solve the problem.

- The first readers-writes solution is implemented as favoring readers and the second solution is implemented as favoring writers.
- In the first one, no readers keep waiting unless a writer has already been granted permission to update the data and programmed as follows.
- In the second one, once a writer is ready to write, it performs its operation as soon as possible and a reader that arrives after a write must wait, even if the writer is already waiting.

You can find the first solution from the lecture as coded below. Based on this solution, **implement the second solution.**

```
/*
* File: assignment7_2.c
* Owner: Yoonseok Yang
* Date: 04.25.2024
* Description: Implement the first readers-writers problem using semaphores
in C
*/

#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

typedef struct {
    sem_t mutex;
    sem_t wlock;
    int readcount;
} rwlock;

typedef struct {
    int data;
    int copy;
    rwlock lock;
} object;

void rwlock_init(rwlock *lock) {
    sem_init(&lock->mutex, 0, 1);
    sem_init(&lock->wlock, 0, 1);
    lock->readcount = 0;
}

void rwlock_deinit(rwlock *lock) {
    sem_destroy(&lock->mutex);
    sem_destroy(&lock->wlock);
}

void acquire_reader_lock(rwlock *lock) {
    sem_wait(&lock->mutex);
    lock->readcount++;
    if(lock->readcount == 1) //if this is the first reader
        sem_wait(&lock->wlock);//wait for a writer to finish or block writers
    sem_post(&lock->mutex);
}

void release_reader_lock(rwlock *lock) {
    sem_wait(&lock->mutex);
    lock->readcount--;
    if(lock->readcount == 0)   //if this is the last reader
        sem_post(&lock->wlock);//unblock any waiting writers
    sem_post(&lock->mutex);
```

```
}

void acquire_writer_lock(rwlock *lock) {
    sem_wait(&lock->wlock);
}

void release_writer_lock(rwlock *lock) {
    sem_post(&lock->wlock);
}

void* reader(void *vargp) {
    object* pobj = (object*)vargp;
    int i;
    for(i = 0; i < 10000; i++) {
        acquire_reader_lock(&pobj->lock);
        int data = pobj->data;
        int copy = pobj->copy;
        release_reader_lock(&pobj->lock);
        printf("R_%d: data: %d, copy: %d\n", i, data, copy);
    }
}

void* writer(void *vargp) {
    object* pobj = (object*)vargp;
    int i;
    for(i = 0; i < 10000; i++) {
        acquire_writer_lock(&pobj->lock);
        int data = pobj->data = i % 10;
        int copy = pobj->copy = pobj->data;
        release_writer_lock(&pobj->lock);
        printf("W_%d: data: %d, copy: %d\n", i, data, copy);
    }
}

int main() {
    pthread_t tid[3];
    object obj;
    obj.data = obj.copy = 0;
    rwlock_init(&obj.lock);

    pthread_create(tid+0, 0, reader, &obj);
    pthread_create(tid+1, 0, reader, &obj);
    pthread_create(tid+2, 0, writer, &obj);

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_join(tid[2], NULL);

    rwlock_deinit(&obj.lock);
}
```

<u>Instructions for the implementation</u>

1. Error handling:
   - Properly handle any error cases and print error messages in the error cases.
2. Comment:
   - Add comments on any of your lines. Describe why you added them in details.
3. Compile:
   - Make your own Makefile to compile.

Documentation

- Document your code thoroughly with comments explaining each section.
- Prepare a report PDF document summarizing everything such as each step or functions that you implemented, challenges faced, and what you've learned during this assignment.

Submission Guidelines:

- Submit the consolidated source code (a zip file) for the entire assignment.
- Include a README file with instructions for compiling and running your program.
- Ensure your code is well-documented with comments.

Assessment Criteria:

Your assignment will be assessed based on the following criteria:

- Submit your own work (80% in points) even though it doesn't execute as expected.
- Proper use of C data types, pointers, arrays, structures, and dynamic memory allocation.
- Efficiency and readability of the code.
- Effective use of memory, functions, and function pointers.
- Proper error handling and validation.
- Utilization of macros for code optimization.
- Utilization of C standard library functions.
- Quality and completeness of documentation.

Important Note: **Plagiarism will not be tolerated, and students are expected to produce their work independently.** Please perform this assignment by yourself and don't copy any solutions from any Generative AI applications like ChatGPT or Bard, your friends or online. If I find any things that imply plagiarism, you will lose the whole points and be reported.