Operating System Lab
Lab #8
Semaphores

Goal:

This lab discusses the semaphore and its application. The expectation is to develop deeper understanding of how to use a semaphore.

Preparation:

Prepare the working environment

- Check out the repo from Github classroom
  - Use `git clone {repo URL}`

- Rebuild Docker image and run the container
  - In a Windows Powershell or MacOS/Liunx terminal, change the current folder to the "lab" folder (the one you check out).
    `cd {lab_folder}`
  - For **Mac** or **Linux**
    - You can execute the shell script "clean_start.sh" if you want to remove old image and create and run a new one.
      `./clean_start.sh`
    - Otherwise, you can just run the following "docker run" command to run a container.
  - For **Windows**
    - You need the following docker commands to run the container from the existing image, *ubuntu_os*.

```
docker run -itd -p8888:8888 --rm --name ubuntu-os-con -v"$PWD/share":/home/app ubuntu_os
```

    - If you do not have the image *ubuntu_os*, you need to rebuild the image

```
docker build -t ubuntu_os .
```

  - To enter the "Bash Shell" of the Ubuntu container for compiling and testing codes

```
docker exec -it ubuntu-os-con /bin/bash
```

- Use VSCode or any text editors for development
  - In the host OS (Windows or MacOS), use terminal to change to the host's lab folder "{check out repo folder}/share" or VSCode to open that host's lab folder. Files you created or modified in the "{check out repo folder}/share" will be shown in docker container's "/home/app" folder.
  - Copy required files from "{check out repo folder}/share" to "{check out repo folder}/answer" and check in for submission.

Part I: Learn POSIX Semaphore

1. Basic

The POSIX system in Linux has a built-in semaphore library. The main functions are shown as follows.

- `sem_init()` - to initialize a semaphore
- `sem_wait()` - to wait on a semaphore
- `sem_post()` - to increment the value of a semaphore (or if it is a binary semaphore, it will release the semaphore)
- `sem_getvalue()` - to get the current value of the semaphore
- `sem_destroy()` - to destroy the semaphore

A semaphore is an object with an integer value that we usually manipulate it with sem_wait() and sem_post(). To use it, we must include the semaphore.h. We can initialize semaphores as follow:

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

We declare a semaphore s and initialize it. From the Linux man page: https://man7.org/linux/man-pages/man3/sem_init.3.html, we know that the setting 0 for the second argument of `sem_init()` indicates that semaphore is shared between threads in the same process. The third argument can be any value. In above example, we pass 1 in as the third argument. After a semaphore is initialized, we can call one of two functions to interact with it, `sem_wait()` or `sem_post()`. The behavior of these two functions are seen in Figure 1.

```
int sem_wait(sem_t *s) {
    decrement the value of semaphore s by one
    wait if value of semaphore s is negative
}

int sem_post(sem_t *s) {
    increment the value of semaphore s by one
    if there are one or more threads waiting, wake one
}
```

Figure 1: Semaphore: definitions of sem_wait and sem_post

We can see that `sem_wait()` will either return right away (because the value of the semaphore was one or higher when we called `sem_wait()`), or it will cause the caller to suspend execution waiting for a subsequent post. Of course, multiple calling threads may call into `sem_wait()`, and thus all be queued waiting to be woken.

We can see that `sem_post()` does not wait for some particular condition to hold like `sem_wait()` does. Rather, it simply increments the value of the semaphore and then, if there is a thread waiting to be woken, wakes one of them up.

2. Binary Semaphores (Locks)

We are now ready to use a semaphore. We utilize binary semaphore as a lock shown in Figure 2.

```
sem_t m;
sem_init(&m, 0, X); //initialize to X; what value should X be?

sem_wait(&m);
// critical section here
sem_post(&m);
```

Figure 2: A binary semaphore implements a lock

You will see that we simply surround the critical section of interest with a `sem_wait()` and `sem_post()` pair.

To-do (Part1):

1. Suppose the value of the current semaphore is -3. How many threads are waiting to enter the critical section? Explain it in the "*answer.md*" under the item "Part1-1"?
2. Take a look at Figure 2 and answer the following questions: Determine what should be X as the initial value of the semaphore m? Explain it in the "*answer.md*" under the item "Part1-2"?
3. Check the unfinished "*binary.c*" code from your answer folder and complete the TODO sections. Answer what the functionality the code has in the "*answer.md*" under the item "Part1-3"?

```
Zem_t s;

//@TODO: add necessary codes to fix this
void *child(void *arg) {
    sleep(4);
    printf("child\n");
    return NULL;
}

int main(int argc, char *argv[]) {
    Zem_init(&s, 0);
    printf("parent: begin\n");
    pthread_t c;
    printf("parent: end\n");
    return 0;
}
```

Figure 3: A parent waiting for its child

4. Semaphores are also useful to order events in a concurrent program. Complete a simple program called "*zemaphore.c*" shown in Figure 3. To do that, check the skeleton code "*zemaphore.c*" in the answer folder and use it in your program. Imagine a thread creates

another thread and then wants to wait for it to complete its execution. The following is the output of the program.

```
parent: begin
child
parent: end
```

Part II: The Producer/Consumer (Bounded Buffer) Problem

Our first attempt at solving the problem introduces two semaphores, empty and full, which the threads will use to indicate when a buffer entry has been emptied or filled, respectively. The code for the put and get functions is shown in Figure 4. The `put()` and `get()` are convenient functions to use buffer. Our attempted solution for the producer and consumer problem is shown in Figure 5.

```c
int *buffer;
int use  = 0;
int fill = 0;

void put(int value) {
    buffer[fill] = value;
    fill++;
    if (fill == max)
    fill = 0;
}

int get() {
    int tmp = buffer[use];
    use++;
    if (use == max)
    use = 0;
    return tmp;
}
```

Figure 4: The put() and get()

In this example, the producer first waits for a buffer to become empty to put data into it, and the consumer similarly waits for a buffer to become filled before using it. Let us first imagine that `max=1` (there is only one buffer in the array) and see if this works.

Consider two threads, a producer and a consumer, and a single CPU. Assume the consumer gets to run first. Thus, the consumer will hit Line C1 in Figure 5 calling `sem_wait(&full)`. Because full was initialized to the value 0, the call will decrement full (to -1), block the consumer, and wait for another thread to call `sem_post()` on full, as desired.

Look at the Line P4. The reason to add `put(-1)` is to make the variable "`tmp`" to receive `-1` and then exit the while loop.

```c
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty); // Line P1
        put(i);           // Line P2
        sem_post(&full);  // Line P3
    }
    put(-1);              // Line P4
    return NULL;
}

void *consumer(void *arg) {
    int i, tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);    // Line C1
        tmp = get();        // Line C2
        sem_post(&empty);   // Line C3
        printf("%lld %d\n", (long long int) arg, tmp);
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    //...
    sem_init(&empty, 0, max); // max are empty
    sem_init(&full, 0, 0);    // 0 are full
    //...
}
```

Figure 5: Adding the full and empty conditions

To-do (Part2):

1. Imagine max =10 for the code shown in Figure 5. Consider multiple producers and consumers. Do we have the race condition? Please put your explanation in "*answer.md*" under the item "Part2-1". The entire program is the "*pc_fig.c*" in the answer folder. You can view it if necessary. Please also fix it if you think the program has issues. If you fix the program, do not need to change the filename.

2. Please start from the skeleton "*pc_works_skeleton.c*" and write a program "*pc_works.c*". "*pc_works.c*" is implemented learning from "*pc_fig.c*" but use the functions in "*common_threads.h*". Therefore, in the "*pc_works.c*", unlike using Linux's semaphore used in "*pc_fig.c*", you need to read the corresponding function in the "*common_thread.h*" to finish your implementation.