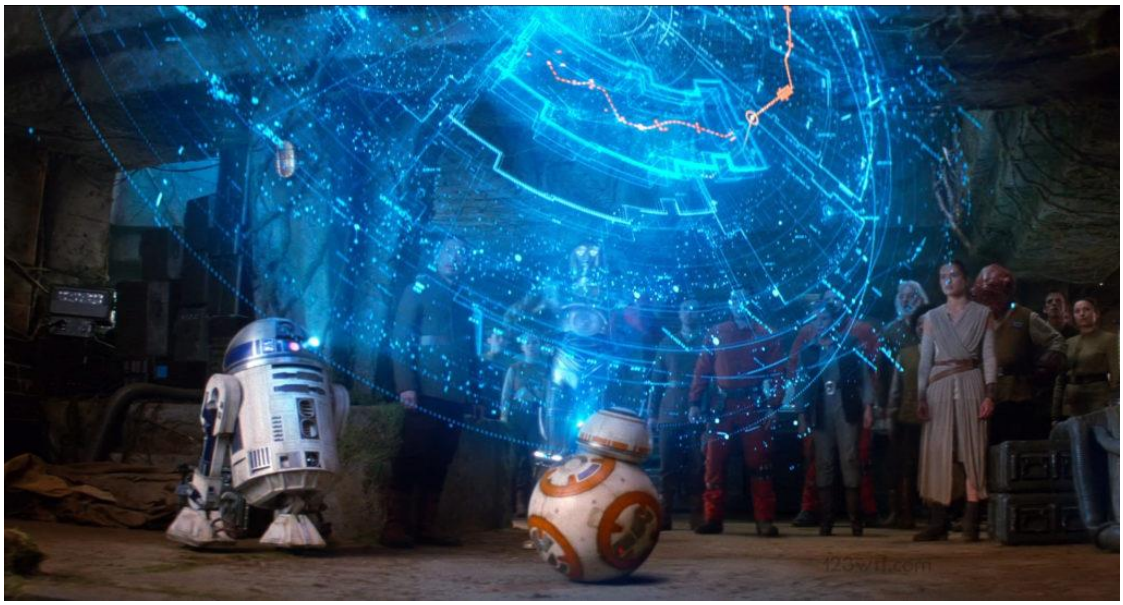
CSE 3100: Lab 8
Instructor: Kriti Bhargava

1 Memory Maps and Process-Level Semaphores

There's always a bigger fish. Qui-Gon Jinn



This lab is meant to give you a chance to get some practice with memory mapping and semaphore APIs.

Shared buffer on steroids.

You are to set up a computation that involves multiple processes and shared maps. The computation is not all that important in this case. What matters is that you set up the collaboration correctly. This computation involves k+1 processes. The orchestration (dance) between the processes proceeds as follows.

Process 1

Process 1 is the "*leader*" and will use two buffers meant to be shared with the other k processes. The **request** buffer is used to share with the "*workers" tasks* to be accomplished (the tasks are trivial and uninteresting here; we provide the code). After placing a fixed number of tasks in the **request** buffer, the *leader* starts consuming responses placed in a second **answer** buffer that is also shared with the worker. The *leader* knows exactly how many answers should come back. It only needs to collect them, print them, and aggregate them for final reporting.

Processes 2 ... k + 1

Processes 2 ... k+1 are "*workers".* They share the two buffers (**request, and answers**) that are used to communicate with the leader. The job of a worker is straightforward. It must consume requests from the **request** buffer until there are no more (the buffer is empty). Each time it picks up a request, it performs a (simple) computation and places the answer in the **answer** buffer. Clearly, a worker terminates when the **request** buffer can no longer deliver any requests.

The ingredients

It goes without saying that the two buffers should be held in shared memory mappings. It is your responsibility to implement those mappings. Thankfully, we do guide you with the API of the buffer. Specifically, buffer.h contains the definition of the data structure.

```
typedef struct Task {
   pid_t worker;
   long long   query;
   long long   answer;
} Task;

typedef struct SBuffer {
   size_t sz;             // max number of tasks in the fixed size buffer
   int enter,leave; nb;   // offsets where to add and retrieve a task
   int nb;                // number of tasks
   size_t mapSize;        // size of the entire mapping in bytes.
   Task  data[0];         // An array of tasks.
} SBuffer;
```

As you realize, a Task is just the "name" of the worker (it's process id that you can get with `getpid`()), the `query`, i.e., the input to the computation and `answer`, the response that it produces.

The shared buffer is equally simple. It is an ADT that holds an array of Tasks whose size is sz. The numerical value `enter` is where a new Task is added to the buffer (a queue) and the numerical value `leave` is where the next Task should be consumed. If you are at the end of the array, you may wrap around to the start. The `data` attribute is an array of `sz` Task. This is a simple C trick where we allocate a memory block substantially larger than what is needed to store an `SBuffer`, store an `SBuffer` instance at the start of the block and use the rest of the block as an array. In this case, we can allocate a block of size `sizeof(SBuffer) + n * sizeof(Task)` and this would be suitable to hold an `SBuffer` instance whose `data` field is an array of `n` Tasks. Of course, we intend to store an SBuffer instance in a memory map created via `mmap`. The APIs for the SBuffer ADT are simply as follows:

```
SBuffer* makeBuffer(void* z,int sz,size_t mapSize);
```

Takes as input the start address of a memory mapping, the number of tasks to hold into the buffer, and the size of the memory mapping (in bytes). Initialize the block pointed to by `z` to hold an SBuffer as defined above.

```
void bufferEnQueue(SBuffer* b,Task t);
```

Adds a task to the buffer (at the end of the queue, i.e., at the enter index). There is no synchronization done here.

```
Task bufferDeQueue(SBuffer* b);
```

Removes a task from the buffer (at the start of the queue, i.e., at the leave index). There is no synchronization done here.

```
SBuffer* setupBuffer(char* zone, int nbTasks);
```

Takes the name of a shared memory object, and a requested number of tasks and creates a memory mapping large enough to accommodate an SBuffer with nbTasks entries. It then initializes the SBuffer with a call to makeBuffer on the obtained memory mapping and returns a pointer to that shared buffer.

```
SBuffer* getBuffer(char* zone, int nbTasks);
```

Given that the name of a shared memory object meant to hold an SBuffer instance with nbTasks, retrieve a pointer to the SBuffer in that memory mapping.

```
void tearDownBuffer(char* zone,SBuffer* b);
```

Deallocate the memory mapping associated with the specified memory zone.

You ought to implement all these functions.

The Leader

We do provide the code of the leader program (prog1.c). It is worth noting that the leader needs to use *semaphores* to synchronize the access to the two shared buffers. Of course, prog1.c will not work until you have the shared buffer implemented. I strongly encourage *all of you* to carefully read and understand the code of the *leader*.

The leader uses two `SBuffer` instances to hold the **requests** and the **answers**. It then defines *four* semaphores (flags) to reason about conditions that must be true for the *leader* to proceed. The narrative below describes each semaphore.

slots

This semaphore starts off with an initial value equal to the number of empty slots available in the **answers** buffer. When it is equal to zero, it means that the **answer** buffer is full of answers and *workers* should not attempt to add more answers here. Each time the *leader* dequeues an answer it should *raise* the slot semaphore to tell a *worker* that there is now an extra empty slot.
You need to figure out how the *worker* should use this flag when manipulating the **answers** buffer.

tasks

This semaphore starts at value 0. It represents the number of tasks that are placed in the **requests** buffer. It is increased by the leader when a task is added to the **request** buffer.
You need to figure out how the *worker* should use this flag when manipulating the **requests** buffer.

asw

This semaphore starts at value 0. It represents the number of tasks that are solved and available in the **answers** buffer. Each time the leader wishes to retrieve an answer (to display it). It must first

lower that semaphore to consume a task. You need to figure out how the *worker* should use this flag when manipulating the **answers** buffer.

lock

This semaphore starts at value 0 and is meant to ensure exclusive access to our toy: the **answer** buffer. Since the *leader* and all the *workers* are manipulating the **answers** buffer concurrently, we need to make sure that all their accesses are coordinated. For instance, we cannot let two *workers* place two finished tasks in the same position in the queue (one would be lost!). Therefore, before doing an enQueue, a *worker* should signal that it wishes to have exclusive access to **answer** *and therefore attempt to lower the semaphore. When the worker is done, it can raise the semaphore to report it is done!* Note how the leader must do the same thing when adding tasks to the queue. Therefore, the call to `bufferDeQueue` is also surrounded by a `sem_wait` and a `sem_post`.

Your mission

It is therefore quite straightforward. You must implement the SBuffer ADT as well as the *worker* program (`prog2.c`). Since the *leader* is provided, it should offer all the necessary guidance to get this done. I cannot emphasize enough that it is essential that you understand what needs to be done before you attempt to write the code. Note that this should work with any number of *workers* from 1 to a (reasonable) value > 1 (e.g., 5).

Note that the different sizes of the **requests** and **answers** buffer are intentional to "test" that the synchronization logic is working.

You can easily test all of this on your VM. Simply run the leader *(prog1)* process on one tab and *worker (prog2) processes* in other tabs. An instance of running this application with 1 leader and 2 worker processes is shown in the screenshot below. Since prog1, sets up the shared object, it must be executed first. The mapping value shows the size of mappings for requests and answers buffer. The number of answer slots are set to 200. The Final answer shows the result of all computations and **must be the same always.**