

## CSIS0230A Principles of Operating Systems(Class A)

### Tutorial 9

#### Reader-Writer Problem

You are given a program (`ReWr.c`) that simulates processes reading and writing some shared objects. However, no synchronization is done, and the program fails when a process tries to read the shared object when another process is busy writing it. The program can be found in `ReWr.c` in the computer in front of you. You don't need to read the complete program, though. Here is all you need to know about the program:

- The program creates `NUM_PROC` (8) processes, all running `do_work()`. The processes are numbered from 0 to 7; this number can be found in the global variable `proccode`.
- `do_work()` loops `NUM_ITERATIONS` (10) iterations, each waits for a while and then performs either a read (calls `do_read()`) or a write (calls `do_write()`). The probability of a write is `WRITE_FRACTION` (0.2).
- The shared object is pointed to by `shared_object`. It is of type **struct** `object_s`, with two fields `values` and `curr_ptr`. The former simply stores a number (starting from 0 and slowly increasing in each write), and the latter points to one of them. (It is designed to do nothing useful but make conflicts very visible). The current value of the object is defined to be `*shared_object->curr_ptr`.

```
struct object_s {
    int* curr_ptr;
    int value;
} *shared_object;
```

- `do_read()` repeatedly wait for a while and fetch the current value of the shared object, expecting that it won't change.

```
int do_read() {
    int i, value;
    value = *shared_object->curr_ptr;
    for (i = 0; i < 10; ++i) {
        readsleep();
        assert(*shared_object->curr_ptr == value);
    }
    return value;
}
```

- `do_write()` read the shared object once, and then make `share_object->curr_ptr` `NULL` so that the current value of the object undefined. Then the process waits for a while, and randomly select a new value for the object.

```
int do_write() {
    int value = *shared_object->curr_ptr; /* SEGV if write in progress */
    shared_object->curr_ptr = NULL; /* make later read and write SEGV */
    writesleep(); /* longer write makes the effect more visible */
    shared_object->curr_ptr = &shared_object->value;
    shared_object->value = value + (rand() % 10);
    return *shared_object->curr_ptr;
}
```

Your task is to use System V semaphores to resolve all conflicts, employing the reader-writer

algorithm of the textbook.

1. Modify **struct** *object\_s* to **add the shared variable** *readcount* needed.
2. Modify the beginning of the *main()* function to **allocate** and **initialize** a **semaphore set** containing the **2 required semaphores**, and **deallocate** it at the end, using *semget()* and *semctl()*.
3. Add code at the beginning and ending of *do\_read()* and *do\_write()* to **perform synchronization**. This should closely resemble the code in the readings. Also, *readcount* should refer to the new variable you created in the shared object.
4. Define the functions *up()* and *down()*, which should call *semop()* to perform the needed semaphore operation. Now test the program.