

CSIS0230A Principles of Operating Systems(Class A)

Tutorial 5

Reader-Writer Problem

In the tutorial, 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 about the program:

- The program creates `NUM_PROC` (8) processes, all running `do_work()`. The processes are numbered from 0 to 7, which 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` (0.1).
- 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 0 to 999, 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 values[1000];
};
struct object_s *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, newvalue;
    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;
    value = *shared_object->curr_ptr;
    shared_object->curr_ptr = NULL;
    writesleep();
    shared_object->curr_ptr = (int*) &shared_object->values[rand() % 1000];
    value = *shared_object->curr_ptr;
    return value;
}
```

Your task is to use SysV 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** the **2 required semaphores**, and **deallocate** it at the end, using *semget* and *semctl*. (Create one set of semaphores to contain both values. The two semaphores needed will be identified by numbers 0 and 1.) Be sure to include the needed header files.
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 textbook, except that you must not use *wait* and *signal* as function names (they are system calls). Perhaps use *WAIT* and *SIGNAL*. Also, *readcount* should be changed to refer to the new variable you created in the shared object.
4. Define the functions *WAIT()* and *SIGNAL()*, which should call *semop()* to perform the needed operation. Now test the program.