Homework 07

CS307-Operating System (D), Chentao Wu, Spring 2020.

Name: 方泓杰(Hongjie Fang) Student ID: 518030910150 Email: galaxies@sjtu.edu.cn

- (7.8) The Linux kernel has a policy that a process cannot hold a spinlock while attempting to acquire a semaphore. Explain why this policy is in place.

  Solution. According to Page 272 in the textbook, spinlocks are often identified as the locking mechanism of choice on multi-processor systems when the lock is to be held for **a short duration** (often less than two context switches). If a process with a spinlock attempt to acquire a semaphore, it may be put into sleep state until the semaphore is available. Therefore, it may hold the spinlock for **a long period of time**, which may be **too long for spinlocks**. Hence, the Linux kernel has a policy that a process cannot hold a spinlock while attempting to acquire a semaphore. □

- (7.11) Discuss the tradeoff between fairness and throughput of operations in the readers-writers problem. Propose a method for solving the readers-writers problem without causing starvation.

  Solution. Here is the explanation of the tradeoff between fairness and throughput of operations in the readers-writers problem.

  - If we allow multiple readers to read data at the same time in the readers-writers problem, it will increase the throughput of operations, but it may cause starvation for writers.
  - If we only allow one reader to read data at the same time, it will be fairer since the statuses of readers and writers are equal, but it will decrease the throughput of operations.

  Here is a fair method for solving the readers-writers problem without causing starvation.

  - Add a timestamp to every reader and writer indicating its arrival time.
  - When a reader arrives and there is no writer in the waiting queue and critical section, it can enter the critical section straightforwardly.
  - When a reader arrives but there is at least one writer in the waiting queue, it should wait in the waiting queue.
  - When a reader arrives but there is a writer in critical section, it should wait in the waiting queue.
  - When a writer arrives and there is no other reader or writer in the critical section, it can enter the critical section straightforwardly.
  - When a writer arrives but there is at least one process (reader or writer) in the critical section, it should wait in the waiting queue.
  - When the last process in the critical section finishes its operation in critical section, then the process with **the longest waiting duration** can enter in the critical section. If this process is a reader, then the next consecutive reader processes can enter in the critical section, too.

  It's a little bit complicated, so here I provide an example.

Example. *There are several processes: reader P_1, reader P_2, writer P_3, writer P_4, reader P_5, reader P_6, writer P_7. Suppose each process need* 1 *unit of time to execute critical section and the processes arrive at time* 0, 0, 0, 1, 2, 2 *and* 3 *respectively.The reader P_1 and P_2 first enter the critical section at time* 0*, then writer P_3 enters at time* 1*. After that, writer P_4 enters at time* 2 *since its waiting duration is the longest in the waiting queue. Then, reader P_5 and P_6 enter the critical section. Finally, writer P_7 enters the critical section.*

It's easy to verify that this method can not cause starvation because the longer waiting duration will give the process a higher priority to be executed. □

- (7.16) There is an implementation of a stack using a linked list. An example of its use is as follows.

```
StackNode *top = NULL;
push(5, &top);
push(10, &top);
push(15, &top);

int value = pop(&top);
value = pop(&top);
value = pop(&top);
```

This program currently has a race condition and is not appropriate for a concurrent environment. Using Pthreads mutex locks (described in Section 7.3.1), fix the race condition.

Solution. The source code after modification is as follows (C++ implementation).

```
# include <iostream>
# include <pthread.h>

using namespace std;

struct StackNode {
    int data;
    StackNode *nxt;           // the next data
    StackNode(int data, StackNode *nxt) : data(data), nxt(nxt) {}
};

StackNode *top;               // the top of stack
pthread_mutex_t mutex;        // lock if there is an operation in push-pop CS.
                              // CS: critical section.

void push(int input_data) {
    pthread_mutex_lock(&mutex);
    StackNode *new_top = new StackNode(input_data, top);
    top = new_top;
    pthread_mutex_unlock(&mutex);
}

int pop() {
    pthread_mutex_lock(&mutex);
```

```
        int output_data = 0;        // default return value 0
        if(top != NULL) {
            output_data = top -> data;
            StackNode *old_top = top;
            top = top -> nxt;
            delete old_top;
        }
        pthread_mutex_unlock(&mutex);
        return output_data;
    }

    int main() {
        pthread_mutex_init(&mutex, NULL);
        top = NULL;

        /* perform push operations and pop operations */
        /* Here is an example */
        push(5);
        push(10);
        push(15);

        int value = pop();
        value = pop();
        value = pop();

        return 0;
    }
```

I design a push-pop lock to prevent racing conditions from occurring. The push-pop lock will be locked only if there is an operation in push-pop critical section. If the push-pop lock is locked, then other processes cannot perform any operations. The push-pop lock guarantees any two operations with `push` or `pop` instructions cannot enter the critical section at the same time. Hence, race condition cannot happen in the modified stack. □