CS2106 Operating Systems
Semester 2 2020/2021

Week of 8th March, 2021
Tutorial 6 **Suggested Solutions**
Synchronization and Deadlocks

1. [**Adapted from AY1920S1 Final – Low Level Implementation of CS**] Multi-core platform X does not support semaphores or mutexes. However, platform **X** supports the following atomic function:

   ```
   bool _sync_bool_compare_and_swap (int* t, int v, int n);
   ```

   The above function atomically compares the value at location pointed by `t` with value `v`. If equal, the function will replace the content of the location with a new value `n`, and return **`1`** (`true`), otherwise return **`0`** (`false`).

   Your task is to implement function `atomic_increment` on platform **X**. Your function should always return the incremented value of referenced location `t`, and be free of race conditions. The use of busy waiting is allowed.

   ```
   int atomic_increment( int* t )
   {
      //your code here
   }
   ```

ANS:

```
int  atomic_increment( int* t )
{
   do {
      int temp = *t;
   } while (!_sync_bool_compare_and_swap(t, temp, temp+1));
   return temp+1;
}
```

2. [**AY 19/20 Midterm – Low Level Implementation of CS]** You are required to implement an intra-process mutual exclusion mechanism (a lock) **using Unix pipes**. Your implementation **should not use mutex** (pthread_mutex) or semaphore (sem), or any other synchronization construct.

   Information to refresh your memory:
   - In multithreaded processes, file descriptors are shared between all threads in a process. If multiple threads simultaneously call `read()` on a file descriptor, only one thread

will be successful in reading any available data up to the buffer size provided, the others will remain blocked until more data is available to be read.

- The read end of a pipe is at index 0, the write end at index 1.
- System calls signatures for `read, write, open, close, pipe` (some might not be needed):

```
int pipe(int pipefd[2]);
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Write your lock implementation below in the spaces provided. Definition of the pipe-based lock (`struct pipelock`) should be complete, but feel free to add any other elements you might need. You need to write code for `lock_init`, `lock_acquire`, and `lock_release`.

ANS:

Line#	Code
1 2 3	**/* Define a pipe-based lock */** struct pipelock { int fd[2]; };
11 12	**/* Initialize lock */** void lock_init(struct pipelock *lock) { **pipe(lock->fd);** **write(lock->fd[1], "a", 1);** **//The first write is meant to initialize the lock such that exactly one thread can acquire the lock.** }
21 22	**/* Function used to acquire lock */** void lock_acquire(struct pipelock *lock) { char c; read(lock->fd[0], &c, 1); //read will block if there is no byte in the pipe. //Closing the reading or writing end of the pipe in a thread will cause closing that end for all threads of the process (shared variable). Also, it will

```
        prevent all other threads to acquire or release the
        lock.
        }
31      /* Release lock */
32      void lock_release(struct pipelock * lock) {

             write(lock->fd[1], "a", 1);

        //Need to write/read exactly one byte to simulate
        increment/decrement by 1 of a semaphore. It might
        work with multiple bytes, but you need to take care
        how many bytes are read/written.



        }
```

Note that the above basically represents simple synchronization through message passing.

3. [**Deadlocks**] We examine the stubborn villagers problem. A village has a long but narrow bridge that does not allow people crossing in opposite directions to pass by each other. All villagers are very stubborn, and will refuse to back off if they meet another person on the bridge coming from the opposite direction.

   a. Explain how the behavior of the villagers can lead to a deadlock.

   b. Analyze the correctness of the following solution and identify the problems, if any.

```
Semaphore sem = 1;

void enter_bridge()
{
   sem.wait();
}

void exit_bridge()
{
   sem.signal();
}
```

   c. Modify the above solution to support multiple people crossing the bridge in the same direction. You are allowed to use a single shared variable and a single semaphore.

   d. What is the problem with solution in (c)?

ANS:

a. Two villagers on different sides of the bridge trying to cross at the same time will lead to a deadlock.

b. The problem with this solution is that it allows only a single villager to cross at a time. A second villager crossing the bridge in the same direction cannot walk behind the first one and instead needs to wait for the first one to exit the bridge.

c. Let's use a single variable `crossing`, whose value is originally `0`. A positive value indicates the number of villagers currently crossing in one direction, and a negative value indicates the number of villagers currently crossing in the other direction.

```
Semaphore mutex=1;
int crossing = 0;

void enter_bridge_direction1()
{
   bool pass=false;
   while(!pass){
      mutex.wait();
      if(crossing>=0){
          crossing++;
         pass=true;
      }
      mutex.signal();
   }
}


void enter_bridge_direction2()
{
   bool pass=false;
   while(!pass){
      mutex.wait();
      if(crossing<=0){
          crossing--;
         pass=true;
      }
      mutex.signal();
   }
}


void exit_bridge_direction1()
{
    mutex.wait();
    crossing--;
    mutex.signal();
}

void exit_bridge_direction2()
{
    mutex.wait();
```

```
    crossing++;
    mutex.signal();
}
```

d. The problem with this solution is that it allows the villagers crossing in one direction to indefinitely starve the villagers crossing in the other direction.

4. [**General Semaphore**] We mentioned that general semaphore (S > 1) can be implemented by using **binary semaphore** (S == 0 or 1). Consider the following attempt:

```
int count =  <initially: any non-negative integer>;
Semaphore mutex = 1;   //binary semaphore
Semaphore queue = 0;    //binary semaphore, for blocking tasks
```

```
GeneralWait() {
    wait( mutex );
    count = count - 1;
    if (count < 0) {
        signal( mutex );
        wait( queue )
    } else {
        signal( mutex );
    }
}
```

```
GeneralSignal() {
    wait( mutex );
    count = count + 1;
    if (count <= 0) {
        signal( queue );
    }
    signal( mutex );
}
```

Note: for ease of discussion, we allow the count to go negative in order to keep track of the number of task blocked on queue.

a. The solution is **very close**, but unfortunately can still have **undefined behavior** in some execution scenarios. Give one such execution scenario to illustrate the issue. (hint: binary semaphore works only when its value S = 0 or S = 1).

b. [Challenge] Correct the attempt. Note that you only need very small changes to the two functions.

ANS:

a. The issue is task interleaving between "`signal(mutex)`" and "`wait(queue)`" in `GeneralWait()` function. Consider the scenario where count is 0, two tasks A and B execute `GeneralWait()`, as task A clears the "`signal(mutex)`", task B gets to executes until the same line. At this point, count is -2. Suppose two other tasks C and D now executes `GeneralSignal()` in turns, both of them will perform `signal(queue)` due to the count -2. Since `queue` is a binary semaphore, the 2nd `signal()` will have undefined behavior (remember that we cannot have S = 2 for binary semaphore).

b. Corrected version:

```
int count =  <any non-negative integer>;
Semaphore mutex = 1;   //binary semaphore
Semaphore queue = 0;    //binary semaphore, for blocking tasks
```

```
GeneralWait() {
    wait( mutex );
    count = count -1;
    if (count < 0) {
        signal(mutex);
        wait(queue)
    }  //else removed
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    } else {    //else added
        signal(mutex);
    }
}
```

Using the same execution scenario in (a), task D will not be able to do the 2nd signal(queue) as the mutex is not unlocked. Either Task A or B can clears the wait(queue), then signal(mutex) allowing task D to proceed. At this point in time, the queue value has settled back to 0 ➔ no undefined signal(queue).

Reference: D. Hemmendinger, "A correct implementation of general semaphores", Operating Systems Review, vol. 22, no. 3 (July, 1988), pp. 42-44.

5. (Discuss if time permits) [**Synchronization Problem – Dining Philosophers**] Our philosophers in the lecture are all left-handed (they pick up the left chopstick first). If we force **one of them** to be a right-hander, i.e. pick up the right chopstick before the left, then it is claimed that the philosophers can eat without explicit synchronization. Do you think this is a **deadlock free solution** to the dining philosopher problem? You can support your claim informally (i.e., no need for a formal proof).

ANS:

The claim is TRUE. Informal argument below.

For ease of discussion, let's refer to the right-hander as **R**.

If **R** grabbed the right fork then managed to grab the left fork THEN
R can eat ➔ not a deadlock.

If **R** grabbed the right fork but the left fork is taken THEN
The left neighbor of R has already gotten both forks ➔ eating ➔ eventually release fork.

If **R** cannot grabbed the right fork THEN
The right neighbor of R has taken its left fork. Worst case scenario: all remaining left-hander all hold on to their left fork. However, the left neighbor of R will be able to take its right fork because R is still trying to get its right fork.