CS2106 Operating Systems

Semester 1 2020/2021

Week of 8th March, 2021

Tutorial 6

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









   }
   ```

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`.

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) { }
21 22	**/* Function used to acquire lock */** void lock_acquire(struct pipelock *lock) { }
31 32	**/* Release lock */** void lock_release(struct pipelock * lock) { }

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)?

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.

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).

Questions for your own exploration

6. Prove that the following solution to the dining philosophers problem is deadlock-free.

```
#define N 5
#define LEFT ((i+N-1)% N)
#define RIGHT ((i+1) % N)
#define THINKING 0
#define HUNGRY 1
#define EATING 2

int state[N];
Semaphore mutex = 1;
Semaphore s[N];

void philosopher( int i ){
    while (TRUE){
     Think( );
     takeChpStcks( i );
     Eat( );
     putChpStcks( i );
    }
}
```

```
void takeChpStcks( i )
{
     wait( mutex );
     state[i] = HUNGRY;
     safeToEat( i );
     signal( mutex );
     wait( s[i] );
}
```

```
void safeToEat( i )
{
    if( (state[i] == HUNGRY) &&
        (state[LEFT] != EATING) &&
        (state[RIGHT] != EATING) ) {
         state[ i ] = EATING;
         signal( s[i] );
    }
}
```

```
void putChpStcks( i )
{
     wait( mutex );
     state[i] = THINKING;
     safeToEat( LEFT );
     safeToEat( RIGHT );
     signal( mutex );
}
```