CS2106 Operating Systems
AY25/26 Semester 1
Tutorial 5
Synchronization

Note: Synchronization is important to both multithreaded and multi-process programs. Hence, we will use the term **task** in this tutorial, i.e. do not distinguish between process and thread.

1. [*Race Conditions*] Consider the following two tasks, *A* and *B*, to be run concurrently and use a shared variable *x*. Assume that:
   - load and store of *x* is atomic
   - *x* is initialized to 0
   - *x* must be loaded into a register before further computations can take place.

Task A	Task B
`x++; x++;`	`x = 2*x;`

   How many **relevant** interleaving scenarios are possible when the two threads are executed? What are all possible values of *x* after both tasks have terminated? Use a stepby-step execution sequence of the above tasks to show all possible results.

2. [Critical Section] Can disabling interrupts avoid race conditions? If yes, would disabling interrupts be a good way of avoiding race conditions? Explain.

3. [**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









}
```

4. [**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 pipebased 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 4	**/* 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) {` `}`