CS2106 Operating Systems
AY25/26 Sem 1
Tutorial 5
Synchronization
SUGGESTED SOLUTIONS

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 step-by-step execution sequence of the above tasks to show all possible results.

ANS:

Load and store instructions to variable *x* are the only instructions whose interleaving can affect the value of *x*. The sequences of such instructions for both processes are shown below:

Task A	Task B
A1: load x A2: store x A3: load x A4: store x	B1: load x B2: store x

Interleaving scenarios can be conveniently represented as a sequence (e.g., A1-A2-A3-A4-B1-B2). How many possible interleaving patterns are there?

Lets start with A1-A2-A3-A4 (the order of instructions cannot change) and see in how many ways we can insert B1 and B2 so that B1 is before B2. We can insert B1 into one of 5 slots (before each A instruction and after the last one). In each of these insertions we have created one new slot for B2, which can now be inserted into one of 6 slots – 30 possible sequences in total. However, in only half of these B1 is before B2, so 5x6/2=15. All of them are shown below, together with the outcomes:

B1-A1-A2-A3-A4-B2 □ $x = 0$
B1-A1-A2-A3-B2-A4 □ $x = 2$
B1-A1-A2-B2-A3-A4 □ $x = 1$
B1-A1- B2-A2-A3-A4 □ $x = 2$
B1-B2-A1-A2-A3-A4 □ $x = 2$

A1-B1-A2-A3-A4-B2 □ $x = 0$
A1-B1-A2-A3-B2-A4 □ $x = 2$
A1-B1-A2-B2-A3-A4 □ $x = 1$
A1-B1-B2-A2-A3-A4 □ $x = 2$

A1-A2-B1-A3-A4-B2 □ $x = 2$
A1-A2-B1-A3-B2-A4 □ $x = 2$
A1-A2-B1-B2-A3-A4 □ $x = 3$

A1-A2-A3-B1-A4-B2 □ $x = 2$
A1-A2-A3-B1-B2-A4 □ $x = 2$

A1-A2-A3-A4-B1-B2 □ $x = 4$

There are five possible outcomes: 0, 1, 2, 3, 4. Note that the outcome would be non-deterministic even if each statement was executed atomically, because x=2*x and x++ are not commutative operations.

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

ANS:

Yes, disabling interrupts and enabling them back is equivalent to acquiring a universal lock and releasing it, respectively. Without interrupts, there will be no quantum-based process switching (because even timer interrupts cannot happen); hence only one process is running. However:

- This will not work in a multi-core/multi-processor environment since another process may enter the critical section while running on a different core.
- User code may not have the privileges needed to disable timer interrupts.
- Disabling timer interrupts means that many scheduling algorithms will not work properly.
- Disabling non-timer interrupts means that high-priority interrupts that may not even share any data with the critical section may be missed.
- Many important wakeup signals are provided by interrupt service routines and these would be missed by the running process. A process can easily block on a semaphore and stay blocked indefinitely, because there is nobody to send a wakeup signal.
- If a program disables interrupts and hangs, the entire system will no longer work since it cannot switch tasks and perform anything else.

Therefore disabling interrupts is not a good choice for the purpose of locking.

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
}
```

ANS:

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

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 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.
}
```

```
/* Release lock */
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.