IE 411: Operating Systems

Practice questions: synchronization

Q1(a) - True or False

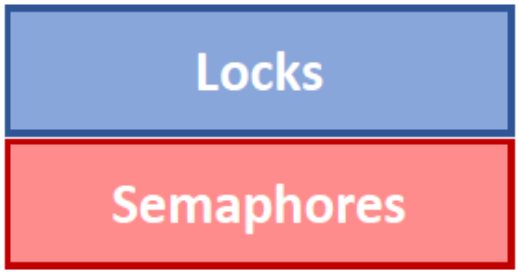
For any problem you can solve with semaphores, you can also solve using condition variables (with a corresponding lock for the condition variable).

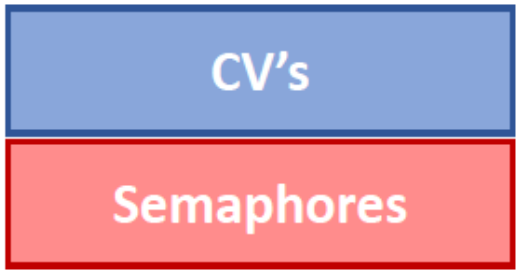
Q1(a) - True or False

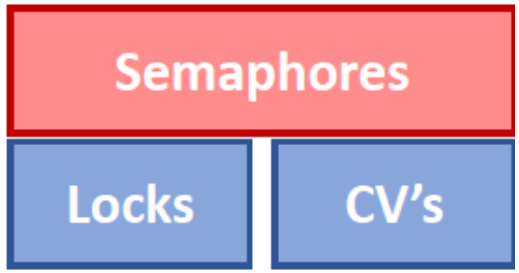
For any problem you can solve with semaphores, you can also solve using condition variables (with a corresponding lock for the condition variable).

TRUE FALSE

Recap







Q1(b) - True or False

A critical section of code is a section of the program that must not be interrupted by the scheduler.

Q1(b) - True or False

A critical section of code is a section of the program that must not be interrupted by the scheduler.

TRUE **FALSE**

Q1(c) - True or False

The function `pthread_cond_signal` never causes its caller to block.

Q1(c) - True or False

The function `pthread_cond_signal` never causes its caller to block.

TRUE FALSE

Q1(d) - True or False

To implement a `thread_join()` operation with a condition variable, the `thread_exit()` code will call `cond_wait()`.

Q1(d) - True or False

To implement a `thread_join()` operation with a condition variable, the `thread_exit()` code will call `cond_wait()`.

TRUE **FALSE**

Q2

```
void thr_exit() {
    mutex_lock(&m);
    pthread_cond_signal(&c);
    mutex_unlock(&m);
}
```

```
void thr_join() {
    mutex_lock(&m);
    pthread_cond_wait(&c, &m);
    mutex_unlock(&m);
}
```

Which is the interleaving for which one of the threads would wait forever?

(a) First, the child runs the code of `thr_exit`. Second, the parent runs `thr_join`.

(b) First, the parent runs the code of `thr_join`. Second, the child runs `thr_exit`.

Answer

Child thread

```
void thr_exit() {
    mutex_lock(&m);
    pthread_cond_signal(&c);
    mutex_unlock(&m);
}
```

Parent thread

```
void thr_join() {
    mutex_lock(&m);
    pthread_cond_wait(&c, &m);
    mutex_unlock(&m);
}
```

- After child finishes the code of `thr_exit()`, parent thread runs the code of `thr_join()`
- When parent calls `pthread_cond_wait()`, it sleeps forever

Q3

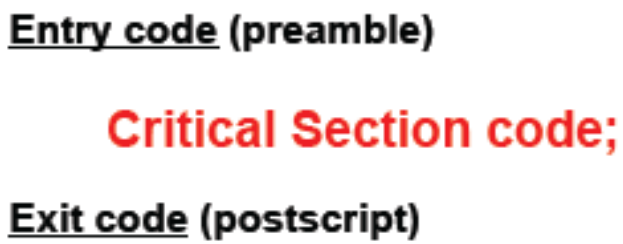
Use an execution sequence to show that it `sem_wait()` is not atomic then mutual exclusion cannot be maintained.

Answer

On the whiteboard

How to prove correctness?

- Critical section: a section of code that only a single process/thread may be executing at a time



- **Proving mutual exclusion**
  - One process in critical section, another process tries to enter $\rightarrow$ Show that second process will block in entry code
  - Two (or more) processes are in the entry code $\rightarrow$ Show that at most one will enter critical section

How to prove correctness?

- **Proving progress**
  - No process in critical section, P1 arrives $\rightarrow$ P1 enters
  - Two (or more) processes are in the entry code $\rightarrow$ Show that at least one will enter critical section

How to prove correctness?

- **Proving progress**
  - No process in critical section, P1 arrives $\rightarrow$ P1 enters
  - Two (or more) processes are in the entry code $\rightarrow$ Show that at least one will enter critical section
- **Proving bounded waiting**
  - One process in critical section, another process is waiting to enter $\rightarrow$ show that if first process exits the critical section and attempts to re-enter, then the waiting process will be get in

Q4

- **Mutual exclusion**
  - One in CS → another that tries to enter fails
  - Two or more try to enter → at most one can succeed
- **Progress**
  - Nobody waiting and one arrives → it will succeed
  - Two or more try to enter → at least one will succeed
- **Bounded wait**
  - One inside and one waiting, 1st exits → 2nd will enter before 1st

Entry code:

```
while (Note) {};
Note = 1;
```

Critical section:

```
foo();
```

Exit code:

```
Note = 0
```

Which of these requirements does this solution fail?

Q5

- **Mutual exclusion**
  - One in CS → another that tries to enter fails
  - Two or more try to enter → at most one can succeed
- **Progress**
  - Nobody waiting and one arrives → it will succeed
  - Two or more try to enter → at least one will succeed
- **Bounded wait**
  - One inside and one waiting, 1st exits → 2nd will enter before 1st

```
flag[2] = {0,0};
```

Entry code:

```
while (flag[them]) {};
flag[me] = 1;
```

Critical section:

```
foo();
```

Exit code:

```
flag[me] = 0;
```

Which of these requirements does this solution fail?

Q6

- **Mutual exclusion**
  - One in CS → another that tries to enter fails
  - Two or more try to enter → at most one can succeed
- **Progress**
  - Nobody waiting and one arrives → it will succeed
  - Two or more try to enter → at least one will succeed
- **Bounded wait**
  - One inside and one waiting, 1st exits → 2nd will enter before 1st

```
flag[2] = {0,0};
```

Entry code:

```
flag[me] = 1;
while (flag[them]) {};
```

Critical section:

```
foo();
```

Exit code:

```
flag[me] = 0;
```

Which of these requirements does this solution fail?

Q7

- **Mutual exclusion**
  - One in CS → another that tries to enter fails
  - Two or more try to enter → at most one can succeed
- **Progress**
  - Nobody waiting and one arrives → it will succeed
  - Two or more try to enter → at least one will succeed
- **Bounded wait**
  - One inside and one waiting, 1st exits → 2nd will enter before 1st

```
turn = 0;
```

Entry code:

```
while (turn != me) {};
```

Critical section:

```
foo();
```

Exit code:

```
turn = them;
```

Which of these requirements does this solution fail?

Q8

- REQ1: At most one thread at a time may be running funcB.
- REQ2: At most two threads at a time may be running any combination of funcA or funcB.

	funcA	funcB
funcA	OK	OK
funcB	OK	NO

Q4

(a) List the semaphores that you will use in your solution. For each semaphore, state what its initial value should be.

(b) Show the semaphore P and V operations that threads should perform before and after each call to funcA and funcB to enforce the synchronization requirements.

```
/* show P, V calls here */

funcA();

/* show P, V calls here */
```

```
/* show P, V calls here */

funcB();

/* show P, V calls here */
```

Answer

- REQ2: At most two threads at a time may be running any combination of funcA or funcB.
- To enforce REQ2, let us use a semaphore `Sem2` with initial value 2

```
P(Sem2);
funcA();
V(Sem2);
```

```
P(Sem2);
funcB();
V(Sem2);
```

Answer

- REQ1: At most one thread at a time may be running funcB.
- To enforce REQ1, let us use a semaphore `Sem1` with initial value 1

```
                                P(Sem1);
P(Sem2);                        P(Sem2);

funcA();                        funcB();

V(Sem2);                        V(Sem2);
                                V(Sem1);
```