CS105 – Computer Systems Spring 2020

Problem Session 4: Synchronization

Wednesday, April 22, 2020

1. For this problem, imagine that you have just been hired by Mother Nature to help her out with the chemical reaction to form water, which she has been struggling with due to synchronization problems. The trick is to get two Hydrogen atoms and one Oxygen atom all together at the same time. The atoms are threads. Each Hydrogen atom invokes a procedure `hReady` when it is ready to react, and each Oxygen atom invokes a procedure `oReady` when it is ready. The procedures should delay until there are at least two Hydrogen atoms and one Oxygen atom present, and then one of the threads must call the procedure `bond`. After the `bond` call, two instances of `hReady` and one of `oReady` should return.

   So far, Mother Nature has come up with two possible solutions. For each approach, determine which of the following is the case:

   (a) The solution is incorrect because **race conditions** are possible.

   (b) The solution is incorrect because it suffers from **starvation** (that is, some thread might wait forever even when the conditions to bond are met)

   (c) The solution is **correct**.

   If the given approach is incorrect, **add** synchronization primitives to make it correct.

   You may assume the semaphore implementation that enforces FIFO order for wakeups, that is the thread waiting longest in `P()` is always the next thread woken up by a call to `V()`.

   (a) Solution 1:

```
sem_t h_wait = sem_init(0);
sem_t o_wait = sem_init(0);
int count = 0;

hReady() {

    count++;

    if(count % 2 == 0) {
        V(o_wait);
    }

    P(h_wait);

    return;
}
```

```
oReady() {

    P(o_wait);

    bond();

    V(h_wait);
    V(h_wait);

    return;
}
```

(b) Solution 2:

```
sem_t h_wait = sem_init(0);
sem_t o_wait = sem_init(0);

hReady(){

    V(o_wait)
    P(h_wait)

    return;
}
```

```
oReady() {

    P(o_wait);
    P(o_wait);

    bond();

    V(h_wait);
    V(h_wait);

    return;
}
```

2. You and a friend have decided to implement an infinite, virtual ping pong game using two threads. One thread writes "Ping!" and another writes "Pong!" The output must strictly alternate—each "Ping!" is immediately followed by a "Pong!", and *vice versa.* The program should satisfy the following properties:

   - "Ping!" goes first. "Ping!" and "Pong!" properly alternate.
   - The game goes on indefinitely. There is no possibility of deadlock.
   - There is no "busy waiting." Neither thread wastes cycles by continuously looping and looking at a variable.

   (a) Your friend has implemented the following version:

```
int ping_count = 0;

void *ping(void* p) {
    while (1){
        if (ping_count == 0) {
            printf("Ping!\n");
            ping_count++;
        }
    }
}

void *pong(void* p) {
    while (1){
        if (ping_count == 1) {
            printf("Pong!\n");
            ping_count--;
        }
    }
}

int main() {
    pthread_t ping_tid, pong_tid;
    pthread_create(&ping_tid, NULL, ping, NULL);
    pthread_create(&pong_tid, NULL, pong, NULL);
    pthread_join(ping_tid, NULL);
    return 0;
}
```

   What is wrong with this implementation?

(b) In the space below, provide a correctly synchronized version by adding global variable(s) of the types `lock` and `cv` and rewriting the funcitons `ping` and `pong`. For simplicity, you may assume that the main function is correct and that the synchronization variables are automatically initialized. You should use the following notation for your synchronization: `mutex_lock(lock)`, `mutex_unlock(lock)`, `cv_wait(cv, lock)`, `cv_signal(cv)`.

```
int ping_count = 0;









void *ping(void* p) {
    while (1) {
```

```
    }
}
```

```
void *pong(void* p) {
    while (1) {
```

```
    }
}
```