CS105 – Computer Systems Spring 2020

Problem Session 4: Synchronization

SOLUTION

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 instance 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:

This solution suffers from a race condition on the shared variable count.

```
sem_t h_wait = sem_init(0);
sem_t o_wait = sem_init(0);
int count = 0;
sem_t lock = sem_init(1);

hReady() {

    P(lock);
    count++;

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

    return;
}
```

```
oReady() {

    P(o_wait);

    bond();

    V(h_wait);
    V(h_wait);

    return;
}
```

(b) Solution 2:

This solution suffers from starvation (e.g., if there are two oxygens that show up before the two hydrogens, and each successfully calls `P(o_wait) once`.

```
sem_t h_wait = sem_init(0);
sem_t o_wait = sem_init(0);
sem_t lock = sem_init(1);

    hReady(){


    V(o_wait)
    P(h_wait)


    return;
}
```

```
oReady() {

    P(lock);
    P(o_wait);
    P(o_wait);
    V(lock);

    bond();


    V(h_wait);
    V(h_wait);


    return;
}
```

2. Use locks and condition variables to synchronize the bounded buffer example we discussed in class.

```
typedef struct {
    int *b;
    int n;
    int front;
    int rear;
    int count;
    Lock lock;
    CV space_avail;
    CV bread_avail;
}


void init(bbuf_t * ptr, int n){
    ptr->b = malloc(n*sizeof(int));
    ptr->n = n;
    ptr->front = 0;
    ptr->rear = 0;
    ptr->count = 0;
    lock_init(ptr->lockt);
    cv_init(ptr->space_avail);
    cv_init(ptr->bread_avail);
}
```

```
void put(bbuf_t * ptr, int val){
    acquire(lock);
    while(ptr->count == ptr->n){
        wait(space_avail);
    }
    ptr->b[ptr->rear] = val;
    ptr->rear = (ptr->rear + 1) % ptr->n;
    ptr->count++;
    signal(bread_avail);
    release(lock);
}

int get(bbuf_t* ptr){
    acquire(lock);
    while(ptr->count == 0){
        wait(bread_avail);
    }
    int val = ptr->b[ptr->front];
    ptr->front = (ptr->front + 1) % ptr->n;
    ptr->count--;
    signal(space_avail);
    release(lock);
}
```