Assignment 2 Notes

Hyungmo Gu

November 4, 2020

1 Read and Writer

- Reader don't modify the data so we can have multiple readers, but only one writer

- Are examples of a common computing problem in concurrency

- Is a part of semaphore problem

2 Product/Consumer

```
producer() {              consumer() {
   while(1) {                while(1) {

      add_to_buf()             remove_from_buf()

   }                        }
}                         }
```

- Is essentially how `pipes()` are implemented

- Has bounded buffer as a shared variable

    - Bounded buffer is also used when piping the output of one program into another

        Example
        `grep foo file.txt | wc -l`

        * `grep`
            · searches the input files for lines containing a match to a given pattern lis
            · when it finds a match in a line, it copies the line to standard output (by default)
        * `wc -l`
            · stands for word count
            · is used to find the number of lines (in this case)
        * `grep` is the producer
        * `wc` is the consumer

- Single buffer producer/consumer solution

– Is to use two different conditial variables

  ∗ Is nice, trouble free and simple

```
1   cond_t   empty, fill;
2   mutex_t mutex;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           Pthread_mutex_lock(&mutex);
8           while (count == 1)
9               Pthread_cond_wait(&empty,  &mutex);
10          put(i);
11          Pthread_cond_signal(&fill);
12          Pthread_mutex_unlock(&mutex);
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          Pthread_mutex_lock(&mutex);
20          while (count == 0)
21              Pthread_cond_wait(&fill,  &mutex);
22          int tmp = get();
23          Pthread_cond_signal(&empty);
24          Pthread_mutex_unlock(&mutex);
25          printf("%d\n", tmp);
26      }
27  }
```

Conditional variable 1

Conditional variable 2

YES

```
1   int loops;
2   cond_t   cond;
3   mutex_t mutex;
4
5   void *producer(void *arg) {
6       int i;
7       for (i = 0; i < loops; i++) {
8           Pthread_mutex_lock(&mutex);            // p1
9           while (count == 1)                     // p2
10              Pthread_cond_wait(&cond,  &mutex);  // p3
11          put(i);                                // p4
12          Pthread_cond_signal(&cond);            // p5
13          Pthread_mutex_unlock(&mutex);          // p6
14      }
15  }
16
17  void *consumer(void *arg) {
18      int i;
19      for (i = 0; i < loops; i++) {
20          Pthread_mutex_lock(&mutex);            // c1
21          while (count == 0)                     // c2
22              Pthread_cond_wait(&cond,  &mutex);  // c3
23          int tmp = get();                       // c4
24          Pthread_cond_signal(&cond);            // c5
25          Pthread_mutex_unlock(&mutex);          // c6
26          printf("%d\n", tmp);
27      }
28  }
```

Same conditional variable

NONO

- The general correct producer/consumer solution

```
1   int buffer[MAX];
2   int fill_ptr = 0;
3   int use_ptr  = 0;
4   int count    = 0;
5
6   void put(int value) {
7       buffer[fill_ptr] = value;
8       fill_ptr = (fill_ptr + 1) % MAX;
9       count++;
10  }
11
12  int get() {
13      int tmp = buffer[use_ptr];
14      use_ptr = (use_ptr + 1) % MAX;
15      count--;
16      return tmp;
17  }
```

Changes made to make solution general

```
1   cond_t empty, fill;
2   mutex_t mutex;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           Pthread_mutex_lock(&mutex);               // p1
8           while (count == MAX)                       // p2
9               Pthread_cond_wait(&empty, &mutex);    // p3
10          put(i);                                    // p4
11          Pthread_cond_signal(&fill);                // p5
12          Pthread_mutex_unlock(&mutex);              // p6
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          Pthread_mutex_lock(&mutex);               // c1
20          while (count == 0)                         // c2
21              Pthread_cond_wait(&fill, &mutex);     // c3
22          int tmp = get();                           // c4
23          Pthread_cond_signal(&empty);               // c5
24          Pthread_mutex_unlock(&mutex);              // c6
25          printf("%d\n", tmp);
26      }
27  }
```

3 Condtional Variable

```
lock_acquire(lock);
while(condition not true) {
    cv_wait(cond, lock);
}
...   // do stuff

cv_signal(cond); //or cv_broadcast(cond)
lock_release(lock);
```

Condition

Conditional variable

- is a queue of waiting threads

- has two operations associated with it:
    1. `cv_wait(struct cv *cv, struct lock *lock)`
        - Is executed when a thread wishes to put itself to sleep
        - Releases lock, waits, re-acquires lock before return
            * Is to prevent race condtions from occuring when a thread is trying to put itself to sleep
    2. `cv_signal(struct cv *cv, struct lock *lock)`
        - Wakes <u>one</u> enqueued thread
    3. `cv_broadcast(struct cv *cv, struct lock *lock)` [from notes]
        - Wakes <u>all</u> enqueued threads
- If no one is waiting, signal or broadcast has no effect
- has rules
    - always use with `while` loops
        * on waking up, tread checks for condition in while loop
        * if condition is true, then thread goes back to sleep
- is always used together with locks

4 Semaphore

```
Wait(Sem){
    while(Sem <= 0) ;
    Sem--;
}
```

```
Signal(Sem){
    Sem++;
}
```

- Was first invented by Dijkstra
- Is abstract data types that provide synchronization
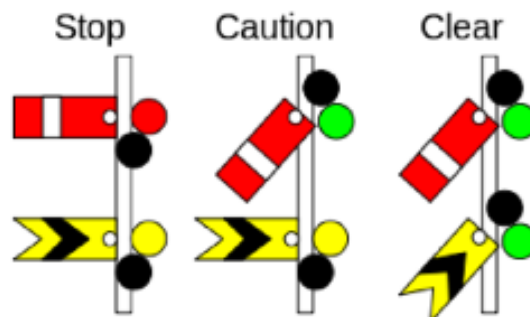
- Has two <u>atomic</u> operations

    - `wait`

        * Is also called P or decrement
        * waits if value of count is negative

    - `signal`

        * increments the variable, unblock a waiting thread if there are any
        * wakes one thread if there are one or more threads waiting

- Has two types

    1. Binary semaphore (count = 0 or 1)
        - Has single access to a resource
        - Can be used like a lock
            * Provides mutual exclusion to a critical section
        - Needs to have initial value of 1
            * Is for decrementation of `count` in `sem_wait()`

```
                                        Need to initialize semaphore to 1
1   sem_t m;
2   sem_init(&m, 0, X); // initialize to X; what should X be?
3
4   sem_wait(&m);
5   // critical section here
6   sem_post(&m);
```

    2. Counting semaphore
        - Resource with many units available
        - Is resource that allows certain kinds of unsynchronized concurrent access (e.g. reading)
        - Mutex has count = 1, counting has count = N

    3. Implementation Tips
        - Implement as few constraints as possible (e.g. how many semaphores do you need to solve the problem?)
        -