

Assignment 2 Notes

Hyungmo Gu

November 5, 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() {
    while(1) {

        add_to_buf()

    }
}

consumer() {
    while(1) {

        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
- 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 conditinal 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 }
18
19 ~
20 cond_t empty, fill;
21 mutex_t mutex;
22
23 void *producer(void *arg) {
24     int i;
25     for (i = 0; i < loops; i++) {
26         Pthread_mutex_lock(&mutex);           // p1
27         while (count == MAX)                  // p2
28             Pthread_cond_wait(&empty, &mutex); // p3
29         put(i);                               // p4
30         Pthread_cond_signal(&fill);           // p5
31         Pthread_mutex_unlock(&mutex);         // p6
32     }
33 }
34
35 void *consumer(void *arg) {
36     int i;
37     for (i = 0; i < loops; i++) {
38         Pthread_mutex_lock(&mutex);           // c1
39         while (count == 0)                    // c2
40             Pthread_cond_wait(&fill, &mutex); // c3
41         int tmp = get();                      // c4
42         Pthread_cond_signal(&empty);          // c5
43         Pthread_mutex_unlock(&mutex);         // c6
44         printf("%d\n", tmp);
45     }
46 }

```

Changes made to make solution general

3 Conditional 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 conditions from occurring when a thread is trying to put itself to sleep
 2. `cv_signal(struct cv *cv, struct lock *lock)`
 - Wakes one enqueued thread
 3. `cv_broadcast(struct cv *cv, struct lock *lock)` [from notes]
 - Wakes all enqueued threads
- If no one is waiting, signal or broadcast has no effect
- has rules
 - always use with while loops
 - * on waking up, thread checks for condition in while loop
 - * if condition is true, then thread goes back to sleep
- is always used together with locks

3.1 Example 1: Read/Writer Problem (Using CV)

```

//number of readers
int readcount = 0;

// Only one writer allowed.
/* No reader while writer is
writing*/

Writer {
    Write;
}

// Multiple readers
/* Readers don't modify any
data */
Reader {
    Read;
}

```

- The problem is about updating and checking the value

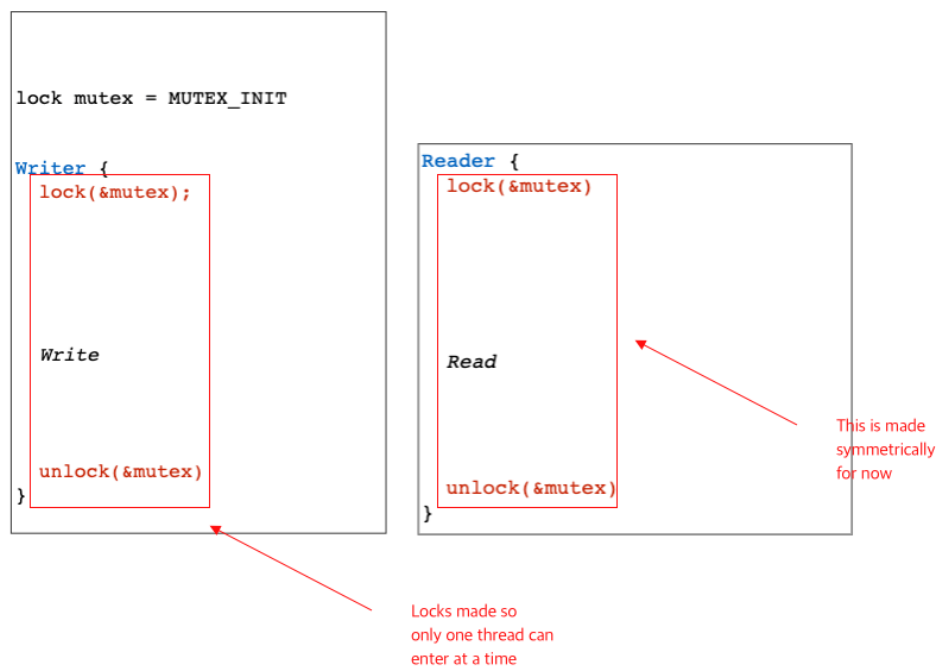
Example

Updating values or reading values in database

- Goals
 - Want only one writer at a time so the writing data is always correct
 - * One reader + One writer or multiple writer creates a problem (race condition)
 - Want many readers at a time because they don't get each other in the way
- Is complex
 - Lots of overhead
- Steps (from lectures)

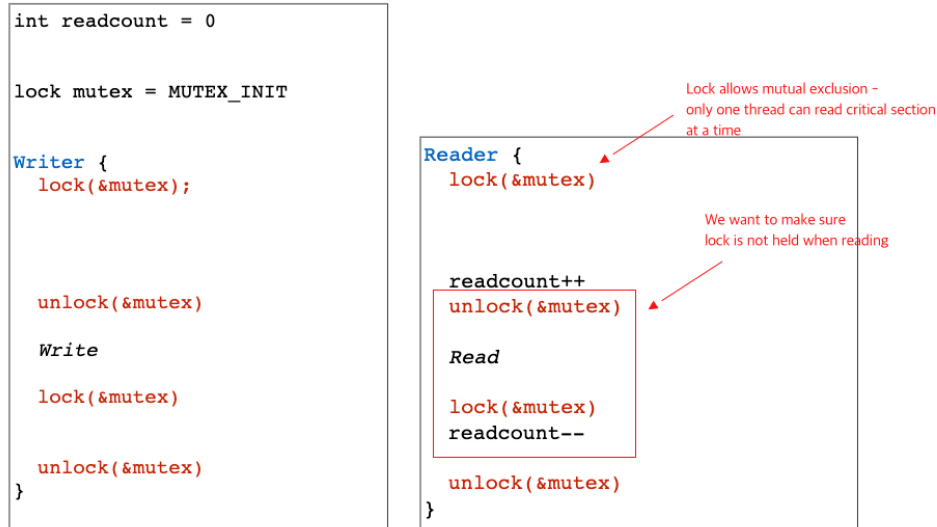
Step 1

- is about making sure only one thread in writer can enter at a time
- symmetric locks in reader are made for now



Step 2

- is about making sure we are not holding the lock while we are reading
- symmetric locks in writer are made for now



Step 3

- is about making sure write doesn't proceed when read is in process

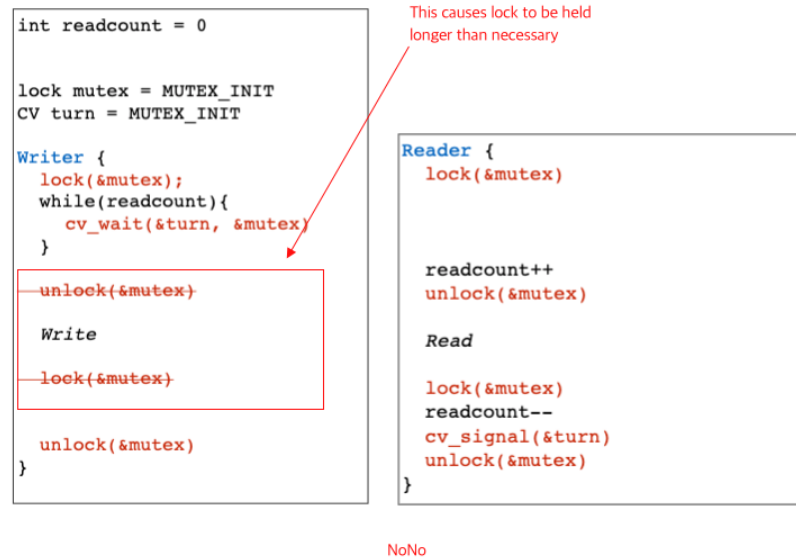


- We can use mutex in reader before critical section because lock in write is released under `cv_wait(...)`

Step 3.5

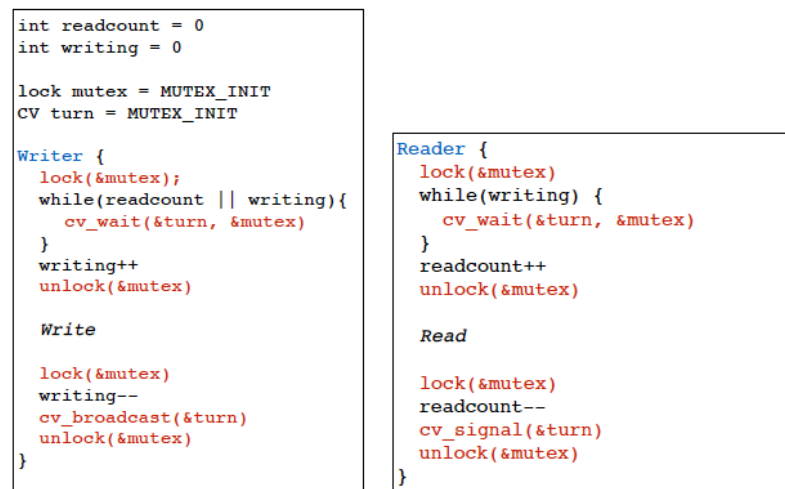
- suppose we try to remove inner lock (to remove extra overhead)
 - * Causes starvation!! not good
 - Reader can keep coming and increase readcount, causing writer to be in blocked state
 - * then we hold lock longer than necessary (Not what we want!!)

- we don't want to hold lock longer than necessary



Step 4

- The purpose is to not hold writing as long as possible
- Solves starvation
 - * Done by adding conditional variable in reader



Step 5

- Think when thread blocked on **turn** wakes up in each reader and writer, can the condition be true?

Step 6

- Think what would happen when we use **signal** or **broadcast**?
 - * New ones in read could bypass the while loop and go into read
 - * Could result in starvation (Nono)
 - * We only want one thread to go in

```
int readcount = 0
int writing = 0

lock mutex = MUTEX_INIT
CV turn = MUTEX_INIT

Writer {
    lock(&mutex);
    while(readcount || writing){
        cv_wait(&turn, &mutex)
    }
    writing++
    unlock(&mutex)

    Write

    lock(&mutex)
    writing--
    cv_broadcast(&turn)
    unlock(&mutex)
}
```

```
Reader {
    lock(&mutex)
    while(writing) {
        cv_wait(&turn, &mutex)
    }
    readcount++
    unlock(&mutex)

    Read

    lock(&mutex)
    readcount--
    cv_signal(&turn)
    unlock(&mutex)
}
```

4 Deadlock

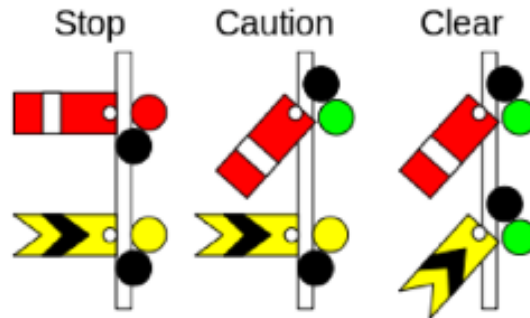
- Conditions
 - Mutual exclusion
 - Hold and wait
 - No preemption
 - Circular wait

5 Semaphore

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

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


- Was first invented by Dijkstra
- Is abstract data types that provide synchronization



- Has two atomic 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()`

```

1  sem_t m;
2  sem_init(&m, 0, X); // initialize to X; what should X be?
3
4  sem_wait(&m); ← Decreases the value of count by 1
5  // critical section here
6  sem_post(&m); ← Increases the value of count by 1

```

Value of Semaphore	Thread 0	Thread 1
1		
1	call <code>sem_wait()</code>	
0	<code>sem_wait()</code> returns	
0	(crit sect)	
0	call <code>sem_post()</code>	
1	<code>sem_post()</code> returns	

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?)

–

5.1 Example 1: Producer/Consumer (Using semaphore)

```

1  void *producer(void *arg) {
2      int i;
3      for (i = 0; i < loops; i++) {
4          sem_wait(&mutex);          // Line P0 (NEW LINE)
5          sem_wait(&empty);          // Line P1
6          put(i);                    // Line P2
7          sem_post(&full);           // Line P3
8          sem_post(&mutex);          // Line P4 (NEW LINE)
9      }
10 }
11
12 void *consumer(void *arg) {
13     int i;
14     for (i = 0; i < loops; i++) {
15         sem_wait(&mutex);          // Line C0 (NEW LINE)
16         sem_wait(&full);           // Line C1
17         int tmp = get();            // Line C2
18         sem_post(&empty);          // Line C3
19         sem_post(&mutex);          // Line C4 (NEW LINE)
20         printf("%d\n", tmp);
21     }
22 }

```