Assignment 2 Notes

Hyungmo Gu

November 14, 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

```
cond_t  empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Conditional variable 1

Conditional variable 2

YES

```
int loops;
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);              // p1
        while (count == 1)                       // p2
            Pthread_cond_wait(&cond, &mutex);    // p3
        put(i);                                  // p4
        Pthread_cond_signal(&cond);              // p5
        Pthread_mutex_unlock(&mutex);            // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);              // c1
        while (count == 0)                       // c2
            Pthread_cond_wait(&cond, &mutex);    // c3
        int tmp = get();                         // c4
        Pthread_cond_signal(&cond);              // c5
        Pthread_mutex_unlock(&mutex);            // c6
        printf("%d\n", tmp);
    }
}
```

Same conditional variable

NONO

- The general correct producer/consumer solution

```
int buffer[MAX];
int fill_ptr = 0;
int use_ptr  = 0;
int count    = 0;

void put(int value) {
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use_ptr];
    use_ptr = (use_ptr + 1) % MAX;
    count--;
    return tmp;
}
```

Changes made to make solution general

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // p1
        while (count == MAX)                  // p2
            Pthread_cond_wait(&empty, &mutex); // p3
        put(i);                               // p4
        Pthread_cond_signal(&fill);           // p5
        Pthread_mutex_unlock(&mutex);         // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // c1
        while (count == 0)                    // c2
            Pthread_cond_wait(&fill, &mutex); // c3
        int tmp = get();                      // c4
        Pthread_cond_signal(&empty);          // c5
        Pthread_mutex_unlock(&mutex);         // c6
        printf("%d\n", tmp);
    }
}
```

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

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

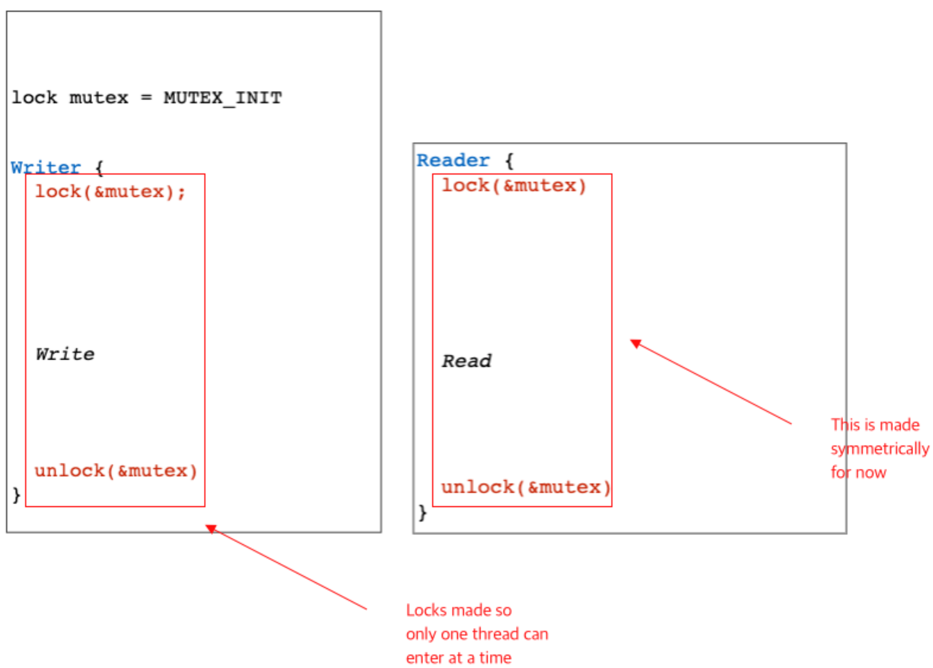
  <u>Example</u>

  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

```
int readcount = 0

lock mutex = MUTEX_INIT

Writer {
  lock(&mutex);


  unlock(&mutex)

  Write

  lock(&mutex)


  unlock(&mutex)
}
```

Lock allows mutual exclusion - only one thread can read critical section at a time

We want to make sure lock is not held when reading

```
Reader {
  lock(&mutex)


  readcount++
  unlock(&mutex)

  Read

  lock(&mutex)
  readcount--

  unlock(&mutex)
}
```

Step 3

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

Uses incrememted value of **readcount** to make sure thread-to-write goes to sleep when reading is in progress

```
int readcount = 0


lock mutex = MUTEX_INIT
CV turn = MUTEX_INIT

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

  Write

  lock(&mutex)


  unlock(&mutex)
}
```

```
Reader {
  lock(&mutex)


  readcount++
  unlock(&mutex)

  Read

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

perform **signal** to wake the thread in **writer**

– 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

```
int readcount = 0

lock mutex = MUTEX_INIT
CV turn = MUTEX_INIT

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

  Write

  lock(&mutex)

  unlock(&mutex)
}
```

This causes lock to be held longer than necessary

```
Reader {
  lock(&mutex)


  readcount++
  unlock(&mutex)

  Read

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

NoNo

Step 4

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

```
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)
}
```

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 happend 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 [Necessary Condition]
    - only one process may use a resource at a time
  - Hold and wait [Necessary Condition]
    - A process may hold allocated resources while awaiting assignment of others
  - No preemption [Necessary Condition]
    - No resource can be forcibly removed from a process holding it
    - You can't steal lock from another thread
  - Circular wait [Sufficient Condition]
    - No resource can be forcibly removed from a process holding it

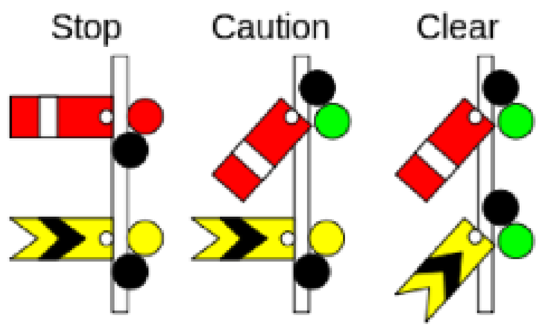
      Example
      - A is holding lock x
      - B is awating for lock x to be released
      - B is holding lock y
      - A is awating for lock y to be released

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 <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

```
sem_t m;
sem_init(&m, 0, X); // initialize to X; what should X be?

sem_wait(&m);
// critical section here
sem_post(&m);
```

sem_wait(&m); — Decreases the value of **count** by 1

sem_post(&m); — Increases the value of **count** by 1

Value of Semaphore	Thread 0	Thread 1
1		
1	call `sem_wait()`	
0	`sem_wait()` returns	
0	`(crit sect)`	
0	call `sem_post()`	
1	`sem_post()` 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)

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);        // Line P0 (NEW LINE)
        sem_wait(&empty);        // Line P1
        put(i);                  // Line P2
        sem_post(&full);         // Line P3
        sem_post(&mutex);        // Line P4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);        // Line C0 (NEW LINE)
        sem_wait(&full);         // Line C1
        int tmp = get();         // Line C2
        sem_post(&empty);        // Line C3
        sem_post(&mutex);        // Line C4 (NEW LINE)
        printf("%d\n", tmp);
    }
}
```

6 Sockets

- port is like an apartment number to the building address

images/notes_1.png

  - program `ssh` is served on port 22
  - program `http` is served on port 80
  - program `https` is served on port 443

- Sometimes connection is made between server and client
  - so data can be sent from server to client and vice versa
  - connection is established using socket

- `stream sockets (TCP)`
  - is connection oriented sockets
  - no loss guarenteed
  - delivery is sent in order
  - `int socket(int domain, int type, int protocol)`

- * `domain` sets the protocol
  - · has two types
  - · `AF_INET`
  - · `PF_INET`
  - · either is fine
- * `type`
  - · many types (e.g. datagram sockets, stream sockets, raw sockets)
  - · we will use `SOCK_STREAM`
- * `protocol`
  - · Here we will use default protocol 0

Example

```
int socket(AF_INET, SOCK_STREAM, 0)
```

7 read

images/notes_3.png

- `read` is blocked until there is something to read or when the other end of the pipe is closed
- When done, read returns and program continues

images/notes_2.png

- Child 1 has nothing to say
  - Since main program waits for child 1 first, the program will block until something happens
- Child 1 has lots to say
  - Since main program waits for child 1 first, nothing will be accepted from the pipe of child 2 by the main program

8 select

-