Week 6.a
CS5600
2/13 2023
https://naizhengtan.github.io/23spring/

0. Last time
1. Mutex
2. Condition variables
3. Semaphores
4. Monitors and standards
5. Advice for concurrent programming

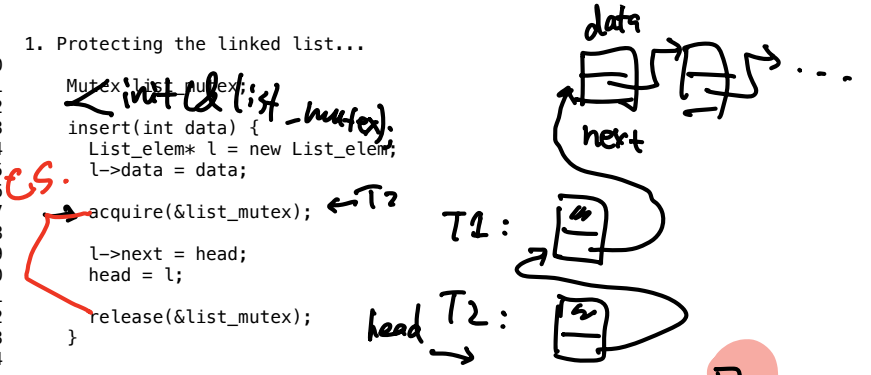
- C.S. → atomicity

- - enter() / lock / acquire
  - C.S
  - leave / unlock / release

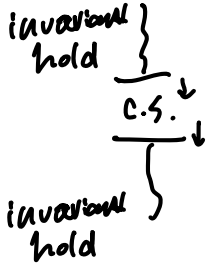
- Mutex

```
pthread_mutex_t m
pthread_mutex_init(mutex_t* m)
{ acquire(mutex_t* m)
  release(mutex_t* m)
```

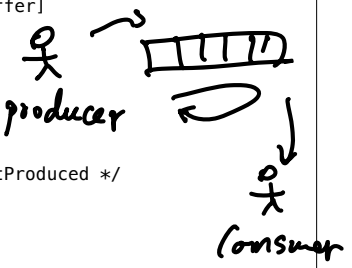
```
1   CS5600 Week05.b
2
3   The handout from the last class gave examples of race conditions.
4   The following panels demonstrate the use of concurrency primitives
5   (mutexes, etc.). We are using concurrency primitives to eliminate
6   race conditions (see items 1 and 2a) and improve scheduling (see item 2b).
7
8
9   1. Protecting the linked list...
10
11      Mutex list_mutex;
12
13      insert(int data) {
14        List_elem* l = new List_elem;
15        l->data = data;
16
17        acquire(&list_mutex);
18
19        l->next = head;
20        head = l;
21
22        release(&list_mutex);
23      }
24
```

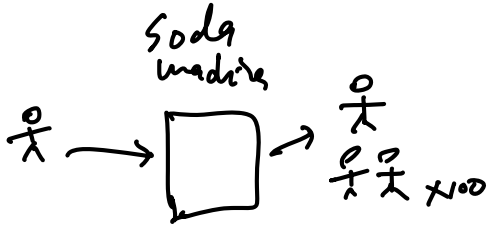


Q: WHY? invariant.




```
25  2. Producer/consumer revisited [also known as bounded buffer]
26
27  2a. Producer/consumer [bounded buffer] with mutexes
28
29      Mutex mutex;
30    void producer (void *ignored) {
31      for (;;) {
32        /* next line produces an item and puts it in nextProduced */
33        nextProduced = means_of_production();
34
35        acquire(&mutex);
36        while (count == BUFFER_SIZE) {
37          release(&mutex);
38          yield(); /* or schedule() */
39          acquire(&mutex);
40        }
41
42        buffer [in] = nextProduced;
43        in = (in + 1) % BUFFER_SIZE;
44        count++;
45        release(&mutex);
46      }
47    }
48
49    void consumer (void *ignored) {
50      for (;;) {
51
52        acquire(&mutex);
53        while (count == 0) {
54          release(&mutex);
55          yield(); /* or schedule() */
56          acquire(&mutex);
57        }
58
59        nextConsumed = buffer[out];
60        out = (out + 1) % BUFFER_SIZE;
61        count--;
62        release(&mutex);
63
64        /* next line abstractly consumes the item */
65        consume_item(nextConsumed);
66      }
67    }
68
69
```

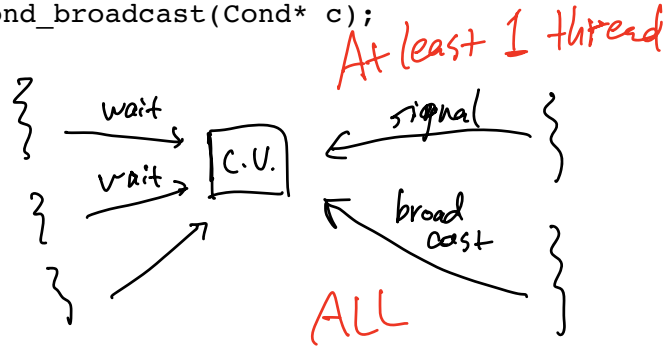




synchronization { ① mutual exclusion (mutex)
② scheduling constraints (C.V.)

cond CV

```
void cond_init (Cond *, ...);
[**] void cond_wait(Cond *cv, Mutex* m);
     void cond_signal(Cond* c);
     void cond_broadcast(Cond* c);
```



① unlock(&m);
wait(&cv)

signal

② lock(&m)
resume

```
70
71  2b. Producer/consumer [bounded buffer] with mutexes and condition variables
72
73      Mutex mutex;
74      Cond nonempty;
75      Cond nonfull;
76
77      void producer (void *ignored) {
78        for (;;) {
79          /* next line produces an item and puts it in nextProduced */
80          nextProduced = means_of_production();
81
82          acquire(&mutex);
83          while (count == BUFFER_SIZE)
84            cond_wait(&nonfull, &mutex);
85
86          buffer [in] = nextProduced;
87          in = (in + 1) % BUFFER_SIZE;
88          count++;
89          cond_signal(&nonempty, &mutex);
90          release(&mutex);
91        }
92      }
93
94      void consumer (void *ignored) {
95        for (;;) {
96
97          acquire(&mutex);
98          while (count == 0)
99            cond_wait(&nonempty, &mutex);
100
101         nextConsumed = buffer[out];
102         out = (out + 1) % BUFFER_SIZE;
103         count--;
104         cond_signal(&nonfull, &mutex);
105         release(&mutex);
106
107         /* next line abstractly consumes the item */
108         consume_item(nextConsumed);
109       }
110     }
111
112
113 Question: why does cond_wait need to both release the mutex and
114 sleep? Why not:
115
116     while (count == BUFFER_SIZE) {
117       release(&mutex);
118       cond_wait(&nonfull);
119       acquire(&mutex);
120     }
121
```


Bounded Buffer

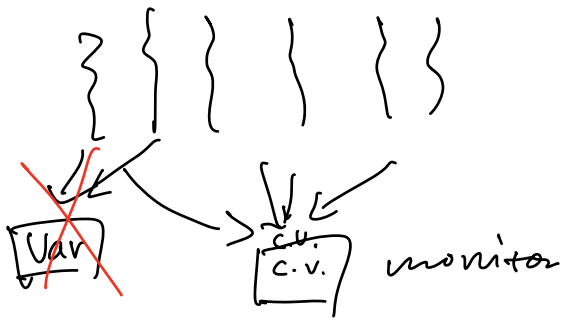
[3 of 6]

Monitor + 6 Rules + 4-step

mutex + C.V.s

obj → foo() {
lock(&m)

unlock(&m)
}



Java.
synchronized

```
1    CS 5600, week 6.a
2
3    The previous handout demonstrated the use of mutexes and condition
4    variables. This handout demonstrates the use of monitors (which combine
5    mutexes and condition variables).
6
7
8    1. The bounded buffer as a monitor
9
10       // This is pseudocode that is inspired by C++.
11       // Don't take it literally.
12
13       class MyBuffer {
14         public:
15           MyBuffer();
16           ~MyBuffer();
17           void Enqueue(Item);
18           Item = Dequeue();
19         private:
20           int count;
21           int in;
22           int out;
23           Item buffer[BUFFER_SIZE];
24           Mutex* mutex;
25           Cond* nonempty;
26           Cond* nonfull;
27       }
28
29       void
30       MyBuffer::MyBuffer()
31       {
32         in = out = count = 0;
33         mutex = new Mutex;
34         nonempty = new Cond;
35         nonfull = new Cond;
36       }
37
38       void
39       MyBuffer::Enqueue(Item item)
40       {
41         mutex.acquire();
42         while (count == BUFFER_SIZE)
43           cond_wait(&nonfull, &mutex);
44
45         buffer[in] = item;
46         in = (in + 1) % BUFFER_SIZE;
47         ++count;
48         cond_signal(&nonempty, &mutex);
49         mutex.release();
50       }
51
52       Item
53       MyBuffer::Dequeue()
54       {
55         mutex.acquire();
56         while (count == 0)
57           cond_wait(&nonempty, &mutex);
58
59         Item ret = buffer[out];
60         out = (out + 1) % BUFFER_SIZE;
61         --count;
62         cond_signal(&nonfull, &mutex);
63         mutex.release();
64         return ret;
65       }
66
```



```
67
68       int main(int, char**)
69       {
70         MyBuffer buf;
71         int dummy;
72         tid1 = thread_create(producer, &buf);
73         tid2 = thread_create(consumer, &buf);
74
75         // never reach this point
76         thread_join(tid1);
77         thread_join(tid2);
78         return -1;
79       }
80
81       void producer(void* buf)
82       {
83         MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
84         for (;;) {
85           /* next line produces an item and puts it in nextProduced */
86           Item nextProduced = means_of_production();
87           sharedbuf->Enqueue(nextProduced);
88         }
89       }
90
91       void consumer(void* buf)
92       {
93         MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
94         for (;;) {
95           Item nextConsumed = sharedbuf->Dequeue();
96
97           /* next line abstractly consumes the item */
98           consume_item(nextConsumed);
99         }
100      }
101
102  Key point: *Threads* (the producer and consumer) are separate from
103  *shared object* (MyBuffer). The synchronization happens in the
104  shared object.
105
```

Four steps

1. Getting started:

   1a. Identify units of concurrency.

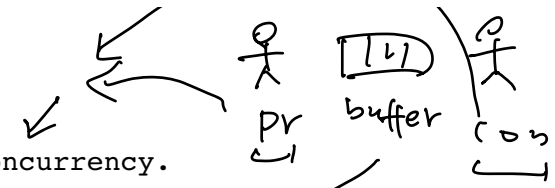
   1b. Identify shared chunks of state.

   1c. Write down the high-level main loop of each thread.

2. Write down the synchronization constraints on the solution.

3. Create a lock or condition variable corresponding to each constraint

4. Write the methods, using locks and condition variables for coordination



① mutual exclusion ⇒ mutex

② if full, then . . ⇒ nonfull

③ . . empty . . . ⇒ nonempty