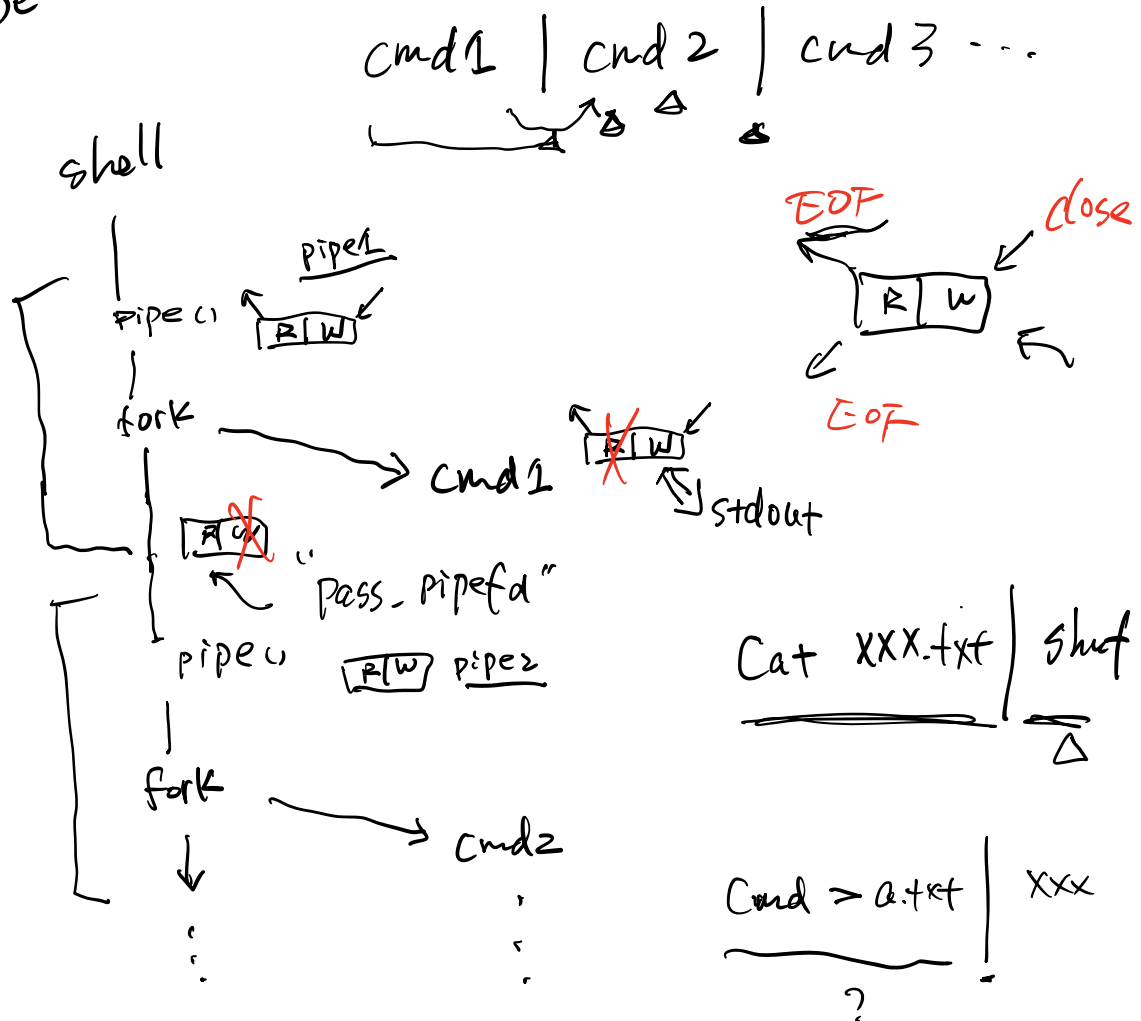


1. Lab2
 2. Condition variables
 3. Semaphores
 4. Monitors and standards
 5. Advice for concurrent programming
-

pipe



mutex \approx lock

C.V.

`cond_init (cond *)` ←
`cond_wait (cond *c, mutex *m)`
 ① unlock mutex (release), ② wait/sleep on c

Signal

③ acquire

mutex, ④ if succ, resume

```

handout w5b
CS5600, Cheng Tan
2/13/22, 3:14 PM
1 CS5600 Week5.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

```

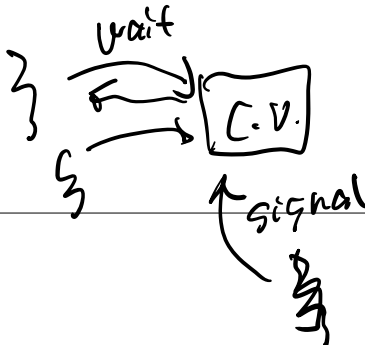
```

handout w5b
CS5600, Cheng Tan
2/13/22, 3:14 PM
25 2. Producer/consumer revisited [also known as bounded buffer]
26
27 2a. Producer/consumer [bounded buffer] with mutexes
28
29     Mutex mutex;
30
31     void producer (void *ignored) {
32         for (;;) {
33             /* next line produces an item and puts it in nextProduced */
34             nextProduced = means_of_production();
35
36             acquire(&mutex);
37             while (count == BUFFER_SIZE) {
38                 release(&mutex);
39                 yield(); /* or schedule() */
40                 acquire(&mutex);
41             }
42
43             buffer [in] = nextProduced;
44             in = (in + 1) % BUFFER_SIZE;
45             count++;
46             release(&mutex);
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

```

Cond - signal (Cond*)

Cond - broadcast (Cond*)




70
71 2b. Producer/consumer [bounded buffer] with mutexes and condition variables

```

72     Mutex mutex;
73     Cond nonempty;
74     Cond nonfull;
75
76     void producer (void *ignored) {
77         for (;;) {
78             /* next line produces an item and puts it in nextProduced */
79             nextProduced = means_of_production();
80
81             acquire(&mutex);
82             while (count == BUFFER_SIZE)
83                 cond_wait(&nonfull, &mutex);
84             buffer [in] = nextProduced;
85             in = (in + 1) % BUFFER_SIZE;
86             count++;
87             cond_signal(&nonempty, &mutex);
88             release(&mutex);
89         }
90     }
91
92     void consumer (void *ignored) {
93         for (;;) {
94             acquire(&mutex);
95             while (count == 0)
96                 cond_wait(&nonempty, &mutex);
97             nextConsumed = buffer[out];
98             out = (out + 1) % BUFFER_SIZE;
99             count--;
100             cond_signal(&nonfull, &mutex);
101             release(&mutex);
102
103             /* next line abstractly consumes the item */
104             consume_item(nextConsumed);
105         }
106     }
107
108     /* next line abstractly consumes the item */
109     consume_item(nextConsumed);
110 }
111
112

```



113 Question: why does cond_wait need to both release the mutex and

114 sleep? Why not:

```

115     while (count == BUFFER_SIZE) {
116         release(&mutex);
117         cond_wait(&nonfull);
118         acquire(&mutex);
119     }
120
121

```

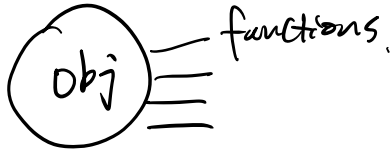
122 wait

FOREVER

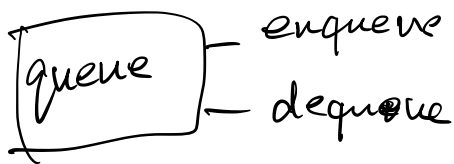
Consumer
acquire(mutex) ...
signal(nonfull)
//finish

X/00

Monitor = mutex + C.U.



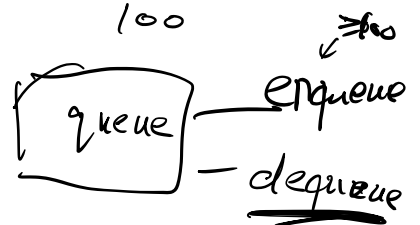
Monitor
6 Rules
4 steps.



DC.U.

↳ limited - queue.

Java. Synchronized



1 C.U.

⇓

2 C.U.

1 CS 5600, week 6.b

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 class MyBuffer {
13     public:
14         MyBuffer();
15         ~MyBuffer();
16         void Enqueue(Item);
17         Item = Dequeue();
18     private:
19         int count;
20         int in;
21         int out;
22         Item buffer[BUFFER_SIZE];
23         Mutex* mutex;
24         Cond* nonempty;
25         Cond* nonfull;
26     };
27
28 void
29 MyBuffer::MyBuffer()
30 {
31     in = out = count = 0;
32     mutex = new Mutex;
33     nonempty = new Cond;
34     nonfull = new Cond;
35 }
36
37 void
38 MyBuffer::Enqueue(Item item)
39 {
40     mutex.acquire();
41     while (count == BUFFER_SIZE)
42         cond_wait(&nonfull, &mutex);
43     buffer[in] = item;
44     in = (in + 1) % BUFFER_SIZE;
45     ++count;
46     cond_signal(&nonempty, &mutex);
47     mutex.release();
48 }
49
50 Item
51 MyBuffer::Dequeue()
52 {
53     mutex.acquire();
54     while (count == 0)
55         cond_wait(&nonempty, &mutex);
56     Item ret = buffer[out];
57     out = (out + 1) % BUFFER_SIZE;
58     --count;
59     cond_signal(&nonfull, &mutex);
60     mutex.release();
61     return ret;
62 }
63
64
65
66

```



monitor

```


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

```

Rules

- acquire/release at beginning/end of methods ↙
- hold lock when doing condition variable operations ↙
- always use "while" to check invariants, not "if" ↙
- ...

`while (Cond)`
`cond_wait (&v, m);` ↙ signal/broadcast ↘



Pr/con

1. Getting started:

1a. Identify units of concurrency.

← Producer Consumer

1b. Identify shared chunks of state.

buffer

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

2.

①. mutual ex → Pr/con

producer

②. buffer full → pr should wait

check buffer
if no full:
enqueue

③. empty → con ...

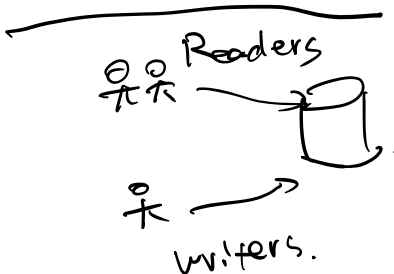
3.

① → mutex

② → nonfull

③ → nonempty

4.



R-R	V
W-W	X
R-W	X

R: access-db1

W: access-db2

R:

check if writer
if no writer:
access

W:

check if w, R
if nobody:
access

```

106 2. This monitor is a model of a database with multiple readers and
107 writers. The high-level goal here is (a) to give a writer exclusive
108 access (a single active writer means there should be no other writers
109 and no readers) while (b) allowing multiple readers. Like the previous
110 example, this one is expressed in pseudocode.
111
112 // assume that these variables are initialized in a constructor
113 state variables:
114 AR = 0; // # active readers
115 AW = 0; // # active writers
116 WR = 0; // # waiting readers
117 WW = 0; // # waiting writers
118
119 Condition okToRead = NIL;
120 Condition okToWrite = NIL;
121 Mutex mutex = FREE;
122
123 Database::read() {
124   startRead(); // first, check self into the system
125   Access Data
126   doneRead();
127 }
128
129 Database::startRead() {
130   acquire(&mutex);
131   while((AW + WW) > 0){
132     WR++;
133     wait(&okToRead, &mutex);
134     WR--;
135   }
136   AR++;
137   release(&mutex);
138 }
139
140 Database::doneRead() {
141   acquire(&mutex);
142   AR--;
143   if (AR == 0 && WW > 0) { // if no other readers still
144     signal(&okToWrite, &mutex);
145   }
146   release(&mutex);
147 }
148
149 Database::write(){ // symmetrical
150   startWrite(); // check in
151   Access Data
152   doneWrite(); // check out
153 }
154
155 Database::startWrite() {
156   acquire(&mutex);
157   while ((AW + AR) > 0) { // check if safe to write.
158     // if any readers or writers, wait
159     WW++;
160     wait(&okToWrite, &mutex);
161     WW--;
162   }
163   AW++;
164   release(&mutex);
165 }
166
167 Database::doneWrite() {
168   acquire(&mutex);
169   AW--;
170   if (WW > 0) {
171     signal(&okToWrite, &mutex); // give priority to writers
172   } else if (WR > 0) {
173     broadcast(&okToRead, &mutex);
174   }
175   release(&mutex);
176 }
177
178 NOTE: what is the starvation problem here?

```

```

179
180 3. Shared locks
181
182 struct sharedlock {
183   int i;
184   Mutex mutex;
185   Cond c;
186 };
187
188 void AcquireExclusive (sharedlock *sl) {
189   acquire(&sl->mutex);
190   while (sl->i) {
191     wait (&sl->c, &sl->mutex);
192   }
193   sl->i = -1;
194   release(&sl->mutex);
195 }
196
197 void AcquireShared (sharedlock *sl) {
198   acquire(&sl->mutex);
199   while (sl->i < 0) {
200     wait (&sl->c, &sl->mutex);
201   }
202   sl->i++;
203   release(&sl->mutex);
204 }
205
206 void ReleaseShared (sharedlock *sl) {
207   acquire(&sl->mutex);
208   if (!--sl->i)
209     signal (&sl->c, &sl->mutex);
210   release(&sl->mutex);
211 }
212
213 void ReleaseExclusive (sharedlock *sl) {
214   acquire(&sl->mutex);
215   sl->i = 0;
216   broadcast (&sl->c, &sl->mutex);
217   release(&sl->mutex);
218 }
219
220 QUESTIONS:
221 A. There is a starvation problem here. What is it? (Readers can keep
222   writers out if there is a steady stream of readers.)
223 B. How could you use these shared locks to write a cleaner version
224   of the code in the prior item? (Though note that the starvation
225   properties would be different.)

```