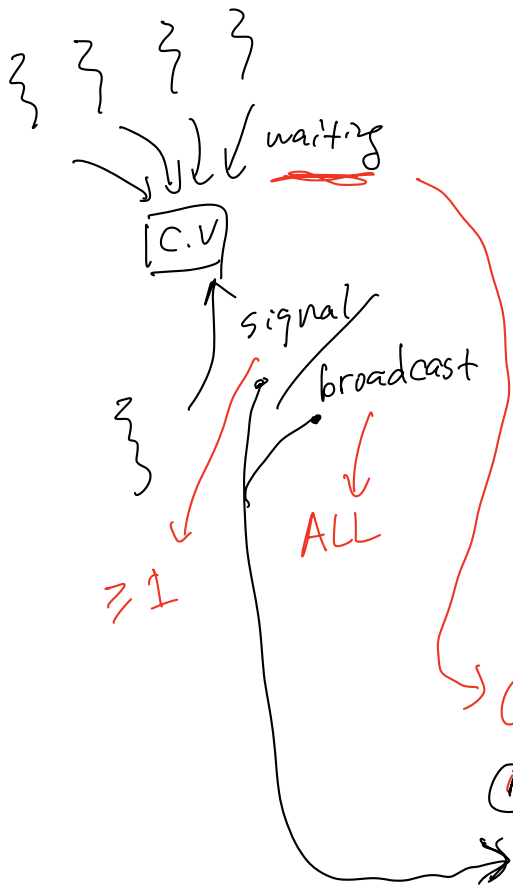


1. Condition variable (cont'd)
2. Monitors and standards
3. Four-step design approach
4. Wrap up concurrency

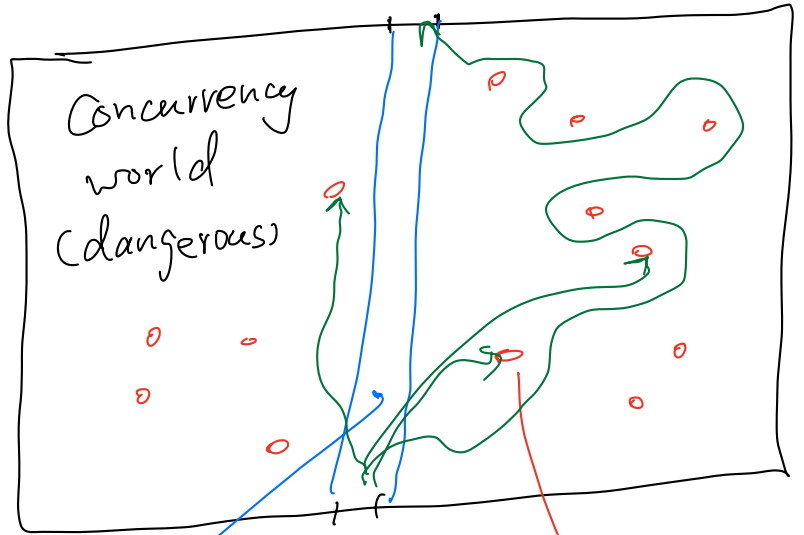
Admin:

- Lab 2
- Survey
- Midterm review

• Condition Variable



mental model of Concurrency



- safe path:
- synchronization primitives
 - Monitor
 - 6 Rules, 4-step approach

```
cond_wait (cond, mutex);  
① unlock mutex; wait on cond  
② lock mutex; if success, resume
```

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

```

103 2b. Producer/consumer [bounded buffer] with mutexes and condition variables
104
105  [ Mutex mutex; ←
106   [ Cond nonempty; ←
107   [ Cond nonfull; ←
108
109  →void producer (void *ignored) {
110     for (;;) {
111         /* next line produces an item and puts it in nextProduced */
112         nextProduced = means_of_production();
113
114         ① acquire(&mutex);
115         while (count == BUFFER_SIZE)
116             cond_wait(&nonfull, &mutex);
117
118         if? buffer [in] = nextProduced;
119         in = (in + 1) % BUFFER_SIZE;
120         count++;
121         →cond_signal(&nonempty, mutex);
122         release(&mutex);
123     }
124 }
125
126 void consumer (void *ignored) {
127     for (;;) {
128
129         acquire(&mutex);
130         while (count == 0)
131             cond_wait(&nonempty, &mutex);
132
133         nextConsumed = buffer[out];
134         out = (out + 1) % BUFFER_SIZE;
135         count--;
136         →cond_signal(&nonfull, mutex);
137         release(&mutex);
138
139         /* next line abstractly consumes the item */
140         consume_item(nextConsumed);
141     }
142 }
143
144

```

Q: WAIT not "if", always "while"??

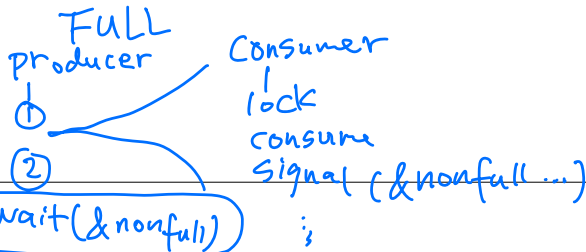
- ① recheck the condition
- ② threads can be waken up "arbitrarily"

Question: why does cond_wait need to both release the mutex and sleep? Why not use

```

148 while (count == BUFFER_SIZE) {
149     ① release(&mutex);
150     ② cond_wait(&nonfull);
151     ③ acquire(&mutex);
152 }

```



2. Monitors and standards

synchronized

↳ an obj in OOP, methods cannot run concurrently



↳ shared states

The six commandments:

1. Always do things the same way
2. Always use monitors (condition variables + locks)
3. Always hold lock when operating on a condition variable
4. Always grab lock at beginning of procedure and release it right before return
5. Always use
`while(predicateOnStateVariables(...) == true/false){
 condition->wait()
}`,
not "if..."
6. (Almost) never sleep()

shared states

problem solving

4-step design approach

6 rules

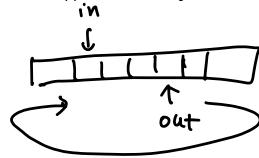
monitor

mutex, C.V.

```

1 CS3650, week 7.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. The buffer should be wrapped as an object.
11 // Don't take it literally.
12
13 // global buffer info
14 int count;
15 int in;
16 int out;
17 Item buffer[BUFFER_SIZE];
18
19 // synchronization variables
20 Mutex* mutex;
21 Cond* nonempty;
22 Cond* nonfull;
23
24 void init()
25 {
26     in = out = count = 0;
27     init(&mutex);
28     init(&non_empty);
29     init(&nonfull);
30 }
31
32 void enqueue(Item item)
33 {
34     acquire(&mutex);
35     while (count == BUFFER_SIZE)
36         cond_wait(&nonfull, &mutex);
37     buffer[in] = item;
38     in = (in + 1) % BUFFER_SIZE;
39     ++count;
40     cond_signal(&nonempty);
41     release(&mutex);
42 }
43
44
45 Item dequeue()
46 {
47     acquire(&mutex);
48     while (count == 0)
49         cond_wait(&nonempty, &mutex);
50
51     Item ret = buffer[out];
52     out = (out + 1) % BUFFER_SIZE;
53     --count;
54     cond_signal(&nonfull);
55     release(&mutex);
56     return ret;
57 }
58

```



buffer \Rightarrow Monitor

```

59 int main(int, char**)
60 {
61     init();
62     tid1 = thread_create(producer, NULL);
63     tid2 = thread_create(consumer, NULL);
64
65     // never reach this point
66     thread_join(tid1);
67     thread_join(tid2);
68     return -1;
69 }
70
71 void producer(void*)
72 {
73     for (;;) {
74         /* next line produces an item and puts it in nextProduced */
75         Item nextProduced = means_of_production();
76         enqueue(nextProduced);
77     }
78 }
79
80 void consumer(void*)
81 {
82     for (;;) {
83         Item nextConsumed = dequeue();
84         /* next line abstractly consumes the item */
85         consume_item(nextConsumed);
86     }
87 }
88
89 Key point: *Threads* (the producer and consumer) are separate from
90 *shared object* (the global buffer). The synchronization happens
91 in the shared object.
92
93

```

Four-step design approach

1. Getting started:

1a. Identify units of concurrency.

producer/consumer

reader/writer

1b. Identify shared chunks of state.

buffer

database.

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

producer:

consumer:

reader:

writer:

loop:

produce item
enqueue buffer

loop:

dequeue buffer
consume item

loop:

check if w?
if so, wait

Read

1 mutex

2 C.V.

loop:

check if w?
if R?

if so, wait

Write

Now, for each object:

1 mutex
2 C.V.

2. Write down the synchronization constraints on the solution.

Identify the type of each constraint: mutual exclusion or scheduling.

For scheduling constraints, ask, "when does a thread wait"?

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

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

A new problem: concurrent database

Consider a database with multiple readers and writers.

The high-level goal here is

(a) to give a writer exclusive access

(a single active writer means there should be no other writers and no readers)

while (b) allowing multiple readers.

94 2. This monitor is a model of a database with multiple readers and
 95 writers. The high-level goal here is (a) to give a writer exclusive
 96 access (a single active writer means there should be no other writers
 97 and no readers) while (b) allowing multiple readers. Like the previous
 98 example, this one is expressed in pseudocode.

```

99
100 // assume that these variables are properly initialized:
101 AR = 0; // active readers
102 AW = 0; // active writers
103 WR = 0; // waiting readers
104 WW = 0; // waiting writers
105
106 Condition okToRead = COND_INITIALIZER;
107 Condition okToWrite = COND_INITIALIZER;
108 Mutex mutex = MUTEX_INITIALIZER;
109
110 void reader() {
111     startRead(); // first, check self into the system
112     Access Data
113     doneRead();
114 }
115
116 void startRead() {
117     acquire(&mutex);
118     while ((AW + WW) > 0) {
119         WR++;
120         wait(&okToRead, &mutex);
121         WR--;
122     }
123     AR++;
124     release(&mutex);
125 }
126
127 void doneRead() {
128     acquire(&mutex);
129     AR--;
130     if (AR == 0 && WW > 0) { // if no other readers still
131         signal(&okToWrite);
132     }
133     release(&mutex);
134 }
135
136 void writer() { // symmetrical
137     startWrite(); // check in
138     Access Data
139     doneWrite(); // check out
140 }
141
142 void startWrite() {
143     acquire(&mutex);
144     while ((AW + AR) > 0) { // check if safe to write.
145         // if any readers or writers, wait
146         WW++;
147         wait(&okToWrite, &mutex);
148         WW--;
149     }
150     AW++;
151     release(&mutex);
152 }
153
154 void doneWrite() {
155     acquire(&mutex);
156     AW--;
157     if (WW > 0) {
158         signal(&okToWrite); // give priority to writers
159     } else if (WR > 0) {
160         broadcast(&okToRead);
161     }
162     release(&mutex);
163 }
164

```

165 NOTE: what is the starvation problem here?

active
 Reader: no writer

a contin- — writers.
 readers will starve