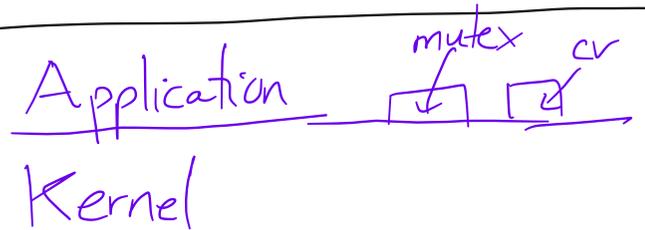


ONE HANDOUT

- 1. Last time
- 2. Condition variables
- 3. Monitors and standards
- 4. Advice
- 5. Practice with concurrent programming

2. Condition variables



A. Motivation

B. API

`cond_init (Cond*, --);`

`cond_wait (Mutex* m, Cond*);`

`cond_signal (Mutex* m, Cond*);`

`cond_broadcast (Mutex* m, Cond*);`

C. Most important point about usage:

MUST "||" + "if" condition

||U>| use while not || when waiting.

D. Other aspects

(1) `cond_wait()` releases mutex and goes into waiting state atomically. Why? Imagine the steps are separate:

producer
⋮
`release(&m);`
`cond_wait(&cv);`
`acquire(&m);`

consumer
`acquire(&m);`
⋮
`cond_signal(&cv);`

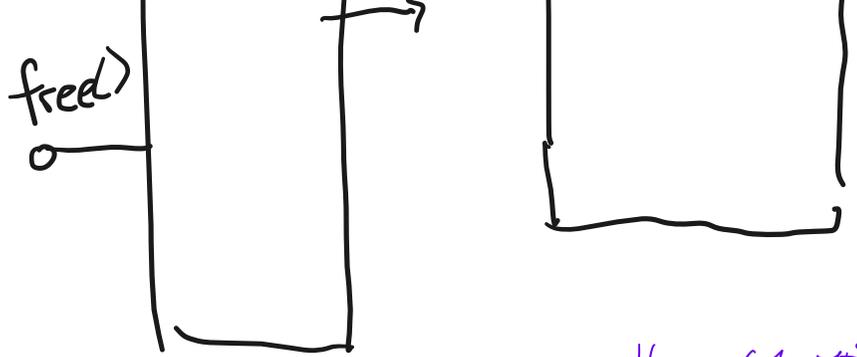
- (2) Can we replace `signal()` with `broadcast()`?
- (3) Can we replace `broadcast()` with `signal()`?

alloc/free example



alloc:

while (not enuf mem)
wait ();



T1 → alloc (1 MB) ✓

T2: alloc (1 KB)

T3: alloc (1 KB)

T4: alloc (5 KB)

free:

~~signal()~~ ?
broadcast()

3. Monitors and standards

Monitor = one mutex + one or more CVs.

The pattern:

```
class Mon {
```

```
private:
```

```
    m, l
```

```
Mon::f()
```

```
{
```

```
    acquire(&m);
```

```

Mutex m;
Cond cv1;
Cond cv2;
:
public:
    f();
    g();
}

Mon::g()
{
    acquire (&m);
    :
    release (&m);
}

```

Example: see handout

Commandments:

Rule: acquire/release at beginning/end of method or function.

Rule: hold lock when doing CV operations

Rule: a thread in wait() must be prepared to

be restarted any time, not just when another thread calls `signal()`;

```
while (not safe to proceed)
wait();
}
```

Rule: don't call `sleep()`;

4. Advice

1. Getting started

1a. identify units of concurrency

1b. identify chunks of state

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

separate threads from objects

2. write down the synchronization constraints, and the kind (mutual exclusion or scheduling)

no producer or consumer accessing shared state at the same time

producer cannot add if buffer full nonfull
" " " " empty nonempty
consumer consume

3. create a lock or CV for each constraint

mutex m

Cond nonfull

Cond non empty

4. write the methods, using the locks and CVs

5. Practice

Example:

- workers interact with a database
- readers never modify
- writers read and modify
- single mutex would be too restrictive
- instead, want:
 - many readers at once OR
 - only one writer (and no readers)

Let's follow the advice:

a. units of concurrency?

b. shared chunks of state?

c. what does main function look like?

read()

check in wait until no writers

access DB()

access - write
check out ... wake waiting writers, if any

write ()

check in ... wait until no one else

access - DB ()

check out ... wake up waiting readers
or writers

2. and 3. synchronization constraints and
synchronization objects

4. write the methods

int AR = 0; // active readers

```
int AR = 0; // active readers  
int AW = 0; // active writers  
int WR = 0; // waiting readers  
int WW = 0; // waiting writers
```

Feb 05, 25 9:50

handout04.txt

Page 1/4

```

1 CS 202
2 Handout 4 (Class 5)
3
4 The handout from the last class gave examples of race conditions. The following
5 panels demonstrate the use of concurrency primitives (mutexes, etc.). We are
6 using concurrency primitives to eliminate race conditions (see items 1
7 and 2a) and improve scheduling (see item 2b).

```

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

```

Feb 05, 25 9:50

handout04.txt

Page 2/4

```

25 2. Producer/consumer revisited [also known as bounded buffer]
26

```

2a. Producer/consumer [bounded buffer] with mutexes

```

27
28     Mutex mutex;
29
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             buffer [in] = nextProduced;
42             in = (in + 1) % BUFFER_SIZE;
43             count++;
44             release(&mutex);
45         }
46     }
47
48     void consumer (void *ignored) {
49         for (;;) {
50             acquire(&mutex);
51             while (count == 0) {
52                 release(&mutex);
53                 yield(); /* or schedule() */
54                 acquire(&mutex);
55             }
56             nextConsumed = buffer[out];
57             out = (out + 1) % BUFFER_SIZE;
58             count--;
59             release(&mutex);
60
61             /* next line abstractly consumes the item */
62             consume_item(nextConsumed);
63         }
64     }
65
66
67
68
69

```

Feb 05, 25 9:50

handout04.txt

Page 3/4

```

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             if ? 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     Question: why does cond_wait need to both release the mutex and
113     sleep? Why not:
114
115     while (count == BUFFER_SIZE) {
116         release(&mutex);
117         cond_wait(&nonfull);
118         acquire(&mutex);
119     }
120
121

```

Feb 05, 25 9:50

handout04.txt

Page 4/4

```

122 2c. Producer/consumer [bounded buffer] with semaphores
123
124     Semaphore mutex(1);          /* mutex initialized to 1 */
125     Semaphore empty(BUFFER_SIZE); /* start with BUFFER_SIZE empty slots */
126     Semaphore full(0);          /* 0 full slots */
127
128     void producer (void *ignored) {
129         for (;;) {
130             /* next line produces an item and puts it in nextProduced */
131             nextProduced = means_of_production();
132
133             /*
134              * next line diminishes the count of empty slots and
135              * waits if there are no empty slots
136              */
137             sem_down(&empty);
138             sem_down(&mutex); /* get exclusive access */
139
140             buffer [in] = nextProduced;
141             in = (in + 1) % BUFFER_SIZE;
142
143             sem_up(&mutex);
144             sem_up(&full); /* we just increased the # of full slots */
145         }
146     }
147
148     void consumer (void *ignored) {
149         for (;;) {
150
151             /*
152              * next line diminishes the count of full slots and
153              * waits if there are no full slots
154              */
155             sem_down(&full);
156             sem_down(&mutex);
157
158             nextConsumed = buffer[out];
159             out = (out + 1) % BUFFER_SIZE;
160
161             sem_up(&mutex);
162             sem_up(&empty); /* one further empty slot */
163
164             /* next line abstractly consumes the item */
165             consume_item(nextConsumed);
166         }
167     }
168
169     Semaphores *can* (not always) lead to elegant solutions (notice
170     that the code above is fewer lines than 2b) but they are much
171     harder to use.
172
173     The fundamental issue is that semaphores make implicit (counts,
174     conditions, etc.) what is probably best left explicit. Moreover,
175     they *also* implement mutual exclusion.
176
177     For this reason, you should not use semaphores. This example is
178     here mainly for completeness and so you know what a semaphore
179     is. But do not code with them. Solutions that use semaphores in
180     this course will receive no credit.

```

Feb 09, 25 23:12

handout05.txt

Page 1/4

```

1 CS 202
2 Handout 5 (Class 6)
3
4 The previous handout demonstrated the use of mutexes and condition
5 variables. This handout demonstrates the use of monitors (which combine
6 mutexes and condition variables).
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

```

Feb 09, 25 23:12

handout05.txt

Page 2/4

```

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

```

Feb 09, 25 23:12

handout05.txt

Page 3/4

```

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

```

Feb 09, 25 23:12

handout05.txt

Page 4/4

```

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.)

```



Operating Systems - Spring 2026, Lecture (SP26:CSCI-UA:202:1:001)

Students

<https://go.blueja.io/ThLCmdvXfUCPi6q0AzasIQ>