

# CS202: Concurrency Tools

Where we are

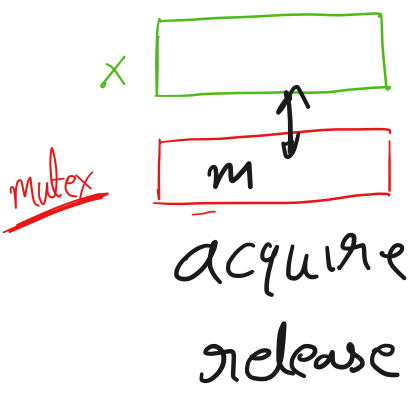
↳ Threads

↳ Multiple simultaneously (logically) executing pieces of code can access the same memory  $\equiv$  CONCURRENCY

→ Hard to reason about what a CONCURRENT program does

↳ NEED TO CONSIDER ALL INTERLEAVINGS

GOAL: FIGURE OUT TOOLS THAT MAKE THIS EASIER.

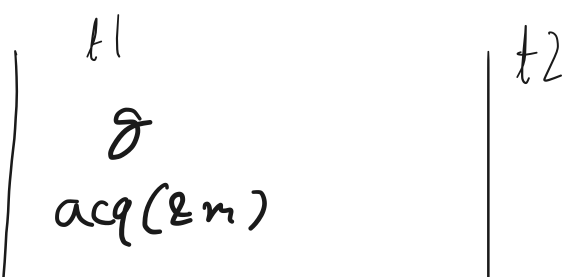
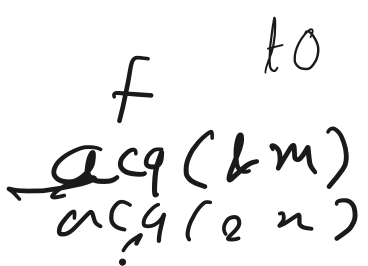


```
f acquire(m)
  x = x + 2;
  x = x * 3;
  release(m)
```

$x = (x) + 2$   
 $7 \cdot 3 = 21$

```
g printf(..., "%d\n", x)
```

```
h x = x - 1;
```



rel (Qm)

mutex: Only one holder at a time

mutex\_init

acquire

release

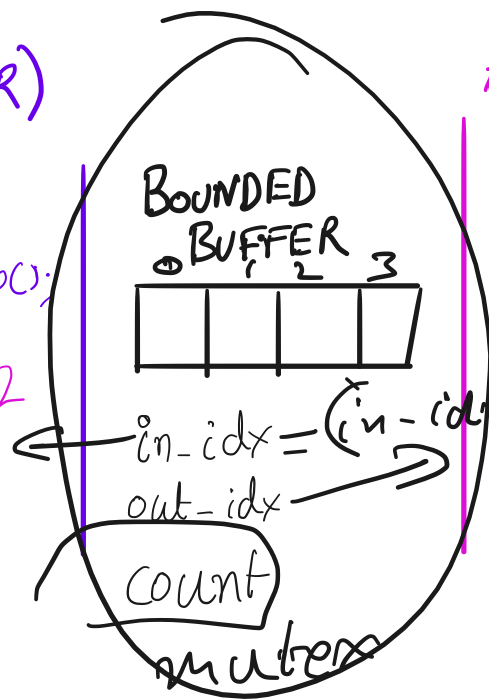
Things to keep in mind when using a mutex

```
acquire(&mutex);  
if (x < 5) {  
    release  
    return;  
}  
else {  
    int y = exp(x, 2);  
    y += 7;  
    x = y;  
} ← release
```

A slightly more complex problem

```

t1 (PRODUCER)
for (...) {
    out = expensive_op();
    // send out to t2
}
    
```

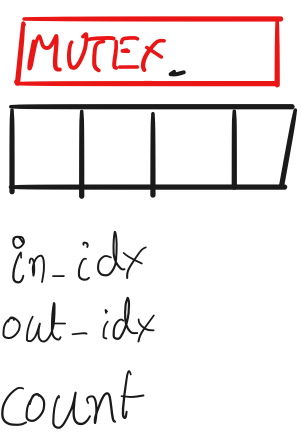


```

t2 (CONSUMER)
while (...) {
    in = // receive from t1
    expensive_op2(in);
}
    
```

Building this with a MUTEX

mutex \*



mutex m;

(Switch to handout 4)

### CONCERNS WITH THE MUTEX VERSION

- CONSUMER UNNECESSARILY ACQUIRES & RELEASES MUTEX

## SOME GENERAL IDEAS ON USING MUTEXES CORRECTLY

- MAKE SURE YOU ALWAYS RELEASE THEM

  - ↳ PUT CRIT. SECTION IN ITS OWN FUNCTION

  - ACQUIRE ON ENTRY

  - SINGLE RETURN

- OR SCOPED LOCKS

## CONDITION VARIABLES

t1 (PRODUCER)

prod(...) {

bounded

SIGNAL THAT ITEMS AVAILABLE

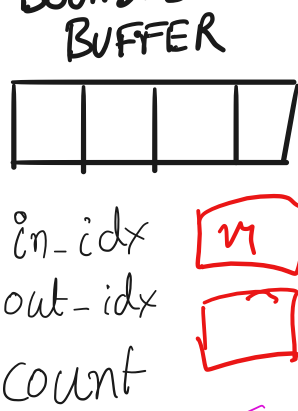
t2 (CONSUMER)

while (...) {

```

out = expensive_opC;
// send out to t2

```



```

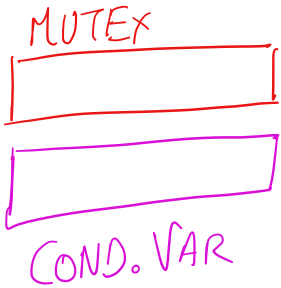
in = // receive from t1
expensive_op2(in);

```

⌋

⌋

SIGNAL THAT SPACE IS AVAILABLE



Condition Variable

cond-init (cv\*, ...)

cond\_wait (cv\*c, mutex\*m)

↳ Caller must hold (have acquired) mutex m.

→ Unlocks/releases m & waits/blocks until c is "notified"

→ locks m before returning.

cond\_signal (cv\*c, mutex\*m)

↳ ① Acquire m

② Notify one thread waiting on c

cond\_broadcast (cv\*c, mutex\*m)

↳ Notify all threads

waiting on  $c$   
→ But what about  $m$ ?

Observe

↳ Mutex  $m$   
protects  $c$

$t1$ : decides to wait on  $c$   
( $count == 0$ )

$t2$ : Does work  
notifies  $c$

$t1$ : wait( $c$ )

X BADXBADX

Let us go look at condition variables in  
action (Handout 4)

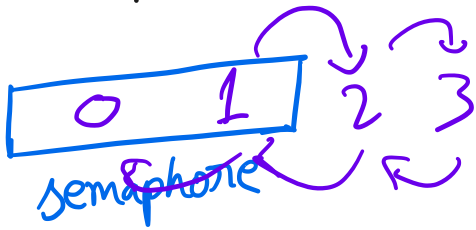
Avoiding Bugs with Cond Var

while loop around wait

~~while (!condition)  
cond\_wait(...)~~

X  
if (!condition)  
cond\_wait(...)

## Semaphores



`sem_init(semaphore*, int)`

`sem_up(semaphore*)`

`sem_down(semaphore*)`

## Handout 4

Please don't use semaphores in this class  
(or generally)

## Monitors

One mutex + one or more cond var

↳ Jointly protect the same data structure.

Look at producer-consumer queue.

## Rules for locks

① Acquire mutex at the beginning of function, release right before return

Single return point is GOOD!!

② Use the same mutex  $m$  for all cond-wait/notify/bcast calls that use a C.V  $c$

③ While loop around cond-wait

④ DON'T CALL SLEEP!!

## Getting started

① Identify units of concurrency

↳ The number of locks they run



↳ threads & what logic they do.

## Producer/consumer...

- ② Identifying the data that is accessed by this concurrent logic
- ③ Write down synchronization constraints  
↳ are they mutual exclusion or scheduling constraints?
- ④ Create mutexes & CVs for each constraint.
- ⑤ Write methods using mutexes & CVs.

Sep 18, 2023 18:00

handout04.txt

Page 1/4

```

1 CS 202, Fall 2023
2 Handout 4
3
4 Handout 3 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.....

```

9
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

```

Sep 18, 2023 18:00

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
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             acquire(&mutex);
52             while (count == 0) {
53                 release(&mutex);
54                 yield(); /* or schedule() */
55                 acquire(&mutex);
56             }
57
58             nextConsumed = buffer[out];
59             out = (out + 1) % BUFFER_SIZE;
60             count--;
61             release(&mutex);
62
63             /* next line abstractly consumes the item */
64             consume_item(nextConsumed);
65         }
66     }
67
68 }
69

```

Handwritten annotations on the code:

- Red arrows pointing to `acquire(&mutex);` and `release(&mutex);` in the producer function.
- Red arrows pointing to `count == BUFFER_SIZE` and `count == 0` in the while loops.
- Red text "not needed" with a red box around the while loops in the consumer function.
- Red text "is it needed" with a red arrow pointing to the `acquire(&mutex);` line in the consumer function.
- Red circles around the `while` loops in both functions.

Sep 18, 2023 18:00

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             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

```

*Handwritten notes:*

- if (count == 0) on
- B
- in
- out
- mutex
- CV nonempty
- CV nonfull

Sep 18, 2023 18:00

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.

```