CS202: Concurrency Tools

Where we are

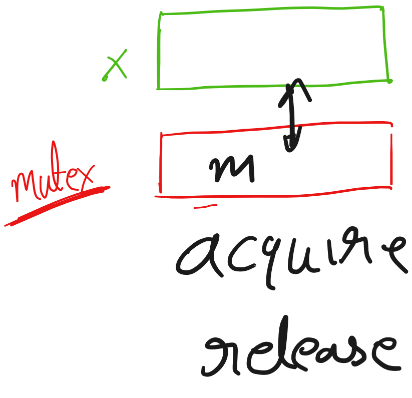
↳ Threads

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

→ Hard to reason about what a CONCURRENT program does

↳ NEED TO CONSIDER ALL INTERLEAVINGS

GOAL: FIGURE OUT TOOLS THAT MAKE THIS EASIER.



acquire

release

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

$x = x + 2$, with 5 for x

7

$7 \cdot 3 = 21$

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

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

t0	t1	t2
f	g	
acq(&m)	acq(&m)	
acq(&n)		
⋮		

rel(&m)

mutex: Only one holder at a time

mutex_init

acquire

release

Things to keep in mind when using a mutex

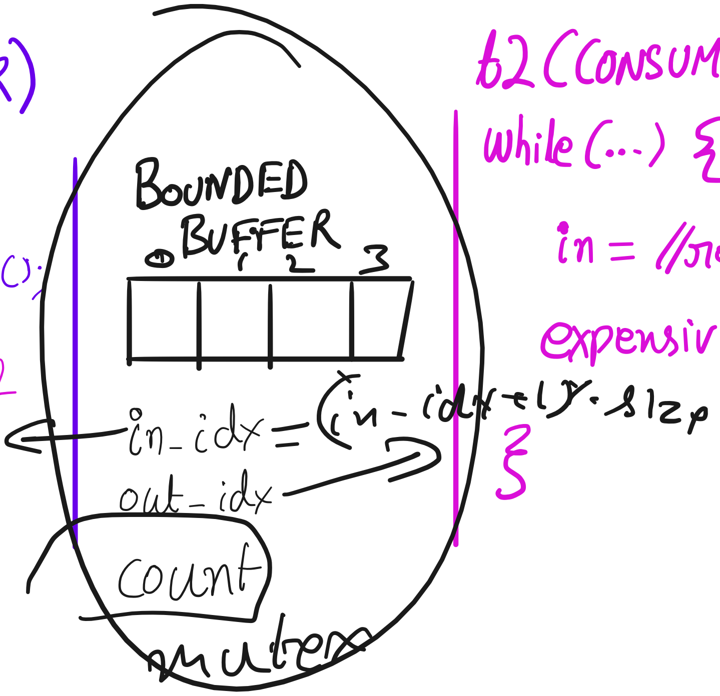
```
acquire(&mx);
if(x<5) { ← release
    return;
} else {
    int y =  exp(x, 2);

    y += 7;
    x = y;
} ← release
```

A slightly more complex problem

t1 (PRODUCER)

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



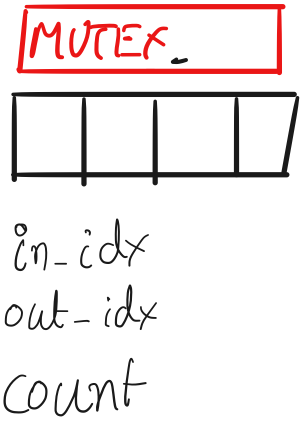
t2 (CONSUMER)

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

Building this with a MUTEX

mutex *

mutex m;



(Switch to handout 4)

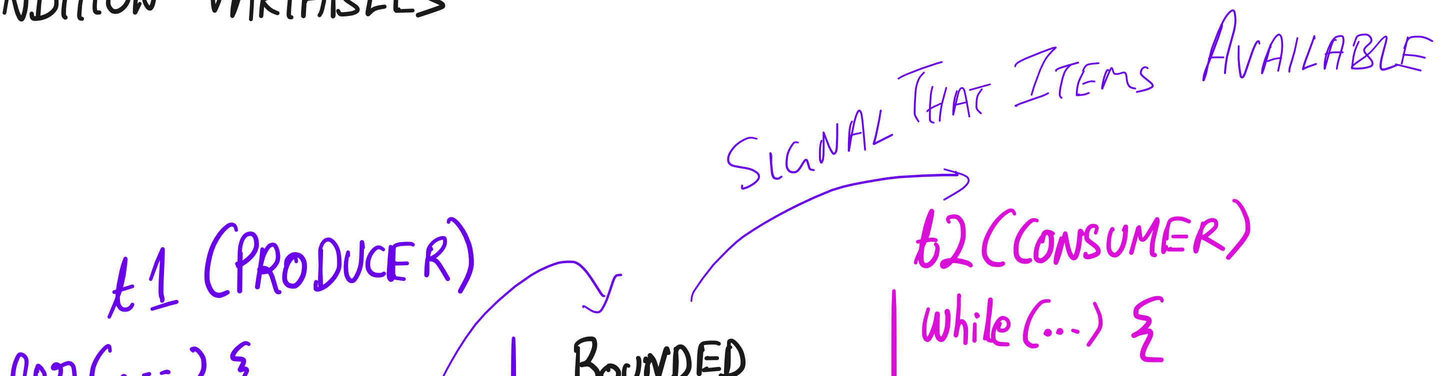
CONCERNS WITH THE MUTEX VERSION

- CONSUMER UNNECESSARILY ACQUIRES & RELEASES MUTEX

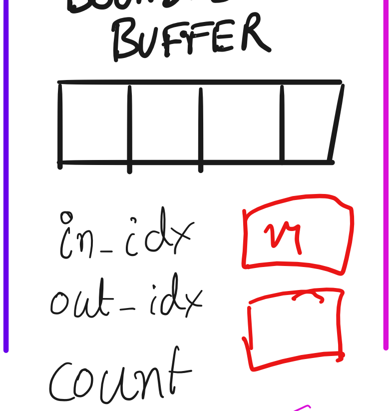
SOME GENERAL IDEAS ON USING MUTEXES CORRECTLY

- MAKE SURE YOU ALWAYS RELEASE THEM
    - → PUT CRIT. SECTION IN ITS OWN FUNCTION
    - → ACQUIRE ON ENTRY
    - → SINGLE RETURN
- OR SCOPED LOCKS

CONDITION VARIABLES



```
    out = expensive_op();
    //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)

× BAD × BAD ×

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

Avoiding Bugs with Cond Var

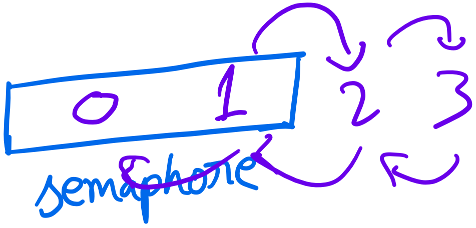
while loop around wait

```
while (!condition)
    cond_wait(..)
```

×

```
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

↳ Threads & what logic they run

↳ threads & what logic they

Producer/consumer/...

② Identify 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.

```
CS 202, Fall 2023
Handout 4

Handout 3 gave examples of race conditions. The following
panels demonstrate the use of concurrency primitives (mutexes, etc.). We are
using concurrency primitives to eliminate race conditions (see items 1
and 2a) and improve scheduling (see item 2b).

1. Protecting the linked list......

        Mutex list_mutex;

        insert(int data) {
            List_elem* l = new List_elem;
            l->data = data;

            acquire(&list_mutex);

            l->next = head;
            head = l;

            release(&list_mutex);
        }

```

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

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

     Mutex mutex;

     void producer (void *ignored) {
         for (;;) {
             /* next line produces an item and puts it in nextProduced */
             nextProduced = means_of_production();

             acquire(&mutex);
             while (count == BUFFER_SIZE) {
                release(&mutex);
                yield(); /* or schedule() */
                acquire(&mutex);
             }

             buffer [in] = nextProduced;
             in = (in + 1) % BUFFER_SIZE;
             count++;
             release(&mutex);
         }
     }

     void consumer (void *ignored) {
         for (;;) {

             acquire(&mutex);
             while (count == 0) {
                release(&mutex);
                yield(); /* or schedule() */
                acquire(&mutex);
             }

             nextConsumed = buffer[out];
             out = (out + 1) % BUFFER_SIZE;
             count--;
             release(&mutex);

             /* next line abstractly consumes the item */
             consume_item(nextConsumed);
         }
     }

```

[illegible] mode.

wait

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

           Mutex mutex;
           Cond nonempty;
           Cond nonfull;

           void producer (void *ignored) {
               for (;;) {
                   /* next line produces an item and puts it in nextProduced */
                   nextProduced = means_of_production();

                   acquire(&mutex);
                   while (count == BUFFER_SIZE)
                      cond_wait(&nonfull, &mutex);

                   buffer [in] = nextProduced;
                   in = (in + 1) % BUFFER_SIZE;
                   count++;
                   cond_signal(&nonempty, &mutex);
                   release(&mutex);
               }
           }

           void consumer (void *ignored) {
               for (;;) {

                   acquire(&mutex);
                   while (count == 0)
                      cond_wait(&nonempty, &mutex);

                   nextConsumed = buffer[out];
                   out = (out + 1) % BUFFER_SIZE;
                   count--;
                   cond_signal(&nonfull, &mutex);
                   release(&mutex);

                   /* next line abstractly consumes the item */
                   consume_item(nextConsumed);
               }
           }
```

if (count == BS)

in

out

1 mutex

CV nonempty

CV nonfull

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

```
            while (count == BUFFER_SIZE)  {
                release(&mutex);
                cond_wait(&nonfull);
                acquire(&mutex);
            }
```

```
      2c.  Producer/consumer [bounded buffer] with semaphores

           Semaphore mutex(1);              /* mutex initialized to 1 */
           Semaphore empty(BUFFER_SIZE);    /* start with BUFFER_SIZE empty slots */
           Semaphore full(0);               /* 0 full slots */

           void producer (void *ignored) {
               for (;;) {
                   /* next line produces an item and puts it in nextProduced */
                   nextProduced = means_of_production();

                  /*
                   * next line diminishes the count of empty slots and
                   * waits if there are no empty slots
                   */
                   sem_down(&empty);
                   sem_down(&mutex);   /* get exclusive access */

                   buffer [in] = nextProduced;
                   in = (in + 1) % BUFFER_SIZE;

                   sem_up(&mutex);
                   sem_up(&full);   /* we just increased the # of full slots */
               }
           }

            void consumer (void *ignored) {
               for (;;) {

                   /*
                    * next line diminishes the count of full slots and
                    * waits if there are no full slots
                    */
                   sem_down(&full);
                   sem_down(&mutex);

                   nextConsumed = buffer[out];
                   out = (out + 1) % BUFFER_SIZE;

                   sem_up(&mutex);
                   sem_up(&empty);   /* one further empty slot */

                   /* next line abstractly consumes the item */
                   consume_item(nextConsumed);
               }
            }
```

Semaphores *can* (not always) lead to elegant solutions (notice that the code above is fewer lines than 2b) but they are much harder to use.

The fundamental issue is that semaphores make implicit (counts, conditions, etc.) what is probably best left explicit. Moreover, they *also* implement mutual exclusion.

For this reason, you should not use semaphores. This example is here mainly for completeness and so you know what a semaphore is. But do not code with them. Solutions that use semaphores in this course will receive no credit.