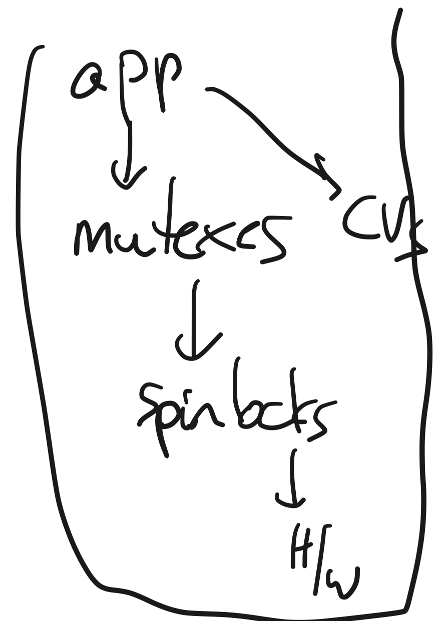
- [x] 1. Last time
- [x] 2. Monitors and standards
- [x] 3. Advice
- [x] 4. Practice w/ concurrent programming
- [ ] 5. (preview) Implementation of locks: spinlocks, mutexes

---

1. Last time: CVs



A. Motivation

B. API

```
cond_init(Cond *, ..);
cond_wait(Cond *, Mutex* m);
cond_signal(Cond *, ...);
cond_broadcast(Cond *, ...);
```

T1: signal(&cv, &m);

T2: if (not_safe())   } producer
      wait(&cv, &m);

buff_full

T3: if (buff_full)
   → producers

C. Important points

- ☑ Must use "while" not "if" when waiting

~~while (_ _ _) { signal(); }~~

- ☑ Cond.wait() releases mutex and goes into waiting state atomically; why?

alternative:

producer:

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

could get an interleaving like this:

consumer:

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

- ☐ Can we replace signal() w/ broadcast()?
- ☐ Can we replace broadcast() w/ signal()?

---

2. Monitors and standards

3. Moni[illegible]

Monitor = <u>one</u> mutex + one or more CVs

The pattern:

```
class Mon {
    private:
        Mutex m;
        Cond  cv1;
        Cond  cv2;
        :
    public:
        f();
        g();
}
```

```
Mon::f()
{
    acquire(&m);
      :
    release(&m);
}

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

Example: see handout

Commandments:

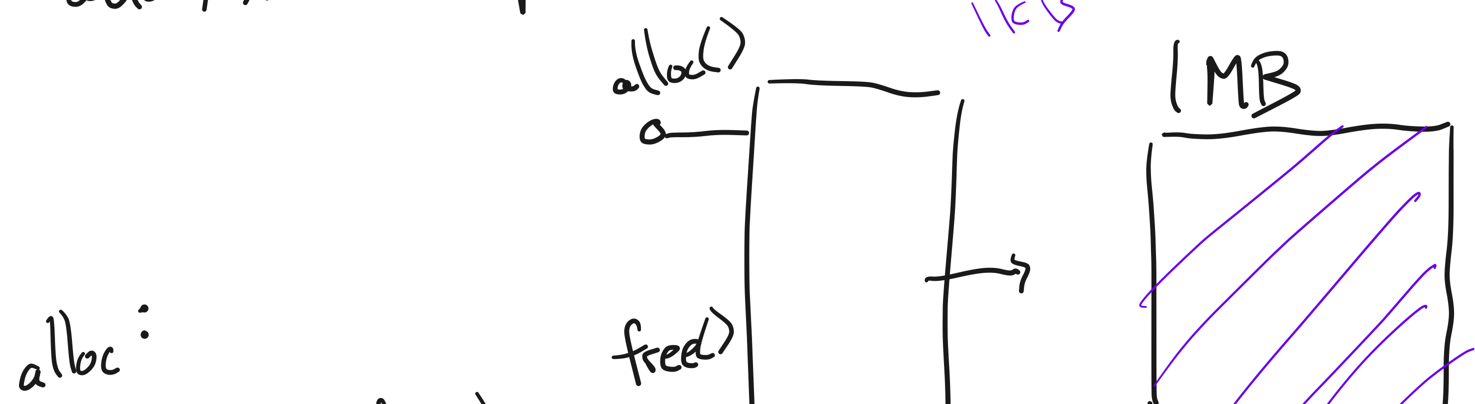
Rule: acquire/release at beginning/end of method or function.

Rule: hold lock when doing CV operations

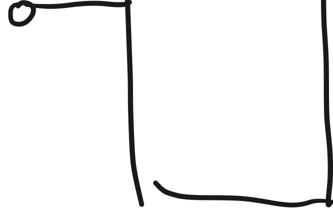
Rule: a thread in wait() must be prepared to be restarted any time, not just when another thread calls signal(); ⟹ while (not safe to proceed) wait();

Rule: don't call sleep();

alloc/free example



while (not enuf mem)
wait ();



T: 1MB

free:
signal() ? broadcast()

```
while (1) {
  it = get_an_item
  put_in_buffer(it);
}
```

3. Advice

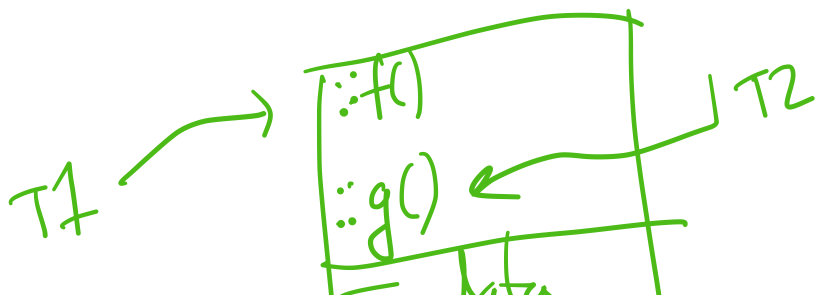
1. Getting started
   - 1a. identify units of concurrency — producers, consumers
   - 1b. identify chunks of state — buffer, count, in, out
   - 1c. write down high-level main loop of each thread

   separate threads from objects

2. write down the synchronization constraints, and the kind (mutual exclusion or scheduling)
   - only one thread modifying shared state
   - producer needs non-full buffer to proceed
   - consumer needs non-empty buffer to proceed

3. Create a lock or CV for each constraint
   - mutex
   - cv1
   - cv2

4. Write the methods, using the locks + CVs



data
members

4. Practice

Example

- workers interact w/ 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? — readers, writers

b. shared chunks of state? — DB, bookkeeping.

c. what does main function look like?

```
read()
    check in .. wait until no writers
    access_DB()
```

check out... wake waiting writers, if any

write()
    checkin ... wait until no one else
    access_DB()
    check out... wake up waiting readers or writers.

2. and 3: synch. constraints and synch. objects

writer can write only if no readers, writers
reader can proceed only if no writer
bookkeeping state (aka shared variables) can be modified by only one thr at a time

startRead()
doneRead()

startWrite();
doneWrite();

4. write the methods

int AR = 0; // active readers
int AW = 0; // active writers

int WR = 0; // waiting readers
int WW = 0; // waiting writers

---

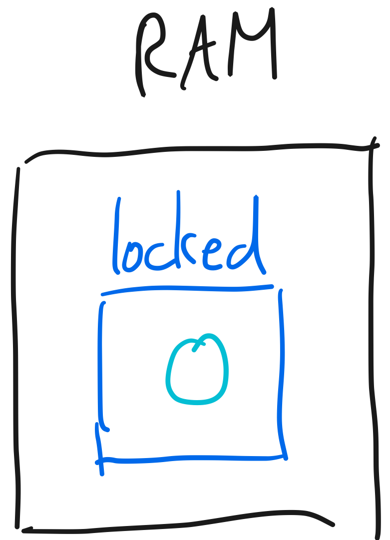
5. Implementation of mutexes

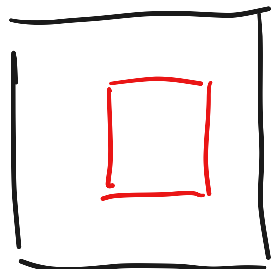
lock()/unlock()?

(a) Peterson ⟶ busy waiting, static bound

(b) disable interrupts

(c) spinlocks





(d) mutexes: spinlock + a queue

- textbook has an implementation
- handout has another

---

ignore below

```
1    [illegible]
2    [illegible]
3
4    Th[illegible] demonstrate[illegible] the [illegible] of mute[illegible] and condition
5    va[illegible]. Th[illegible] demonstrates the use of [illegible] (which combine
6    m[illegible] and [illegible] variables).
7
8    [illegible] bounded buffer [illegible]
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* notempty;
26           Cond* nofull;
27       }
28
29       void
30       MyBuffer::MyBuffer()
31       {
32           in = out = count = 0;
33           mutex = new Mutex;
34           notempty = new Cond[illegible]
35           nofull = new Cond[illegible]
36       }
37
38       void
39       MyBuffer::Enqueue(Item item)
40       {
41           mutex.acquire();
42           while (count == BUFFER_SIZE)
43               cond_wait(nofull, &mutex);
44
45           buffer[in] = item;
46           in = (in + 1) % BUFFER_SIZE
47           ++count;
48           cond_signal(notempty, &mutex);
49           mutex.release()
50       }
51
52       Item
53       MyBuffer::Dequeue()
54       {
55           mutex.acquire();
56           while (count == 0)
57               cond_wait(notempty, &mutex);
58
59           Item ret = buffer[out];
60           out = (out + 1) % BUFFER_SIZE;
61           --count;
62           cond_signal(nofull, &mutex);
63           mutex.release()
64           return ret;
65       }
66
67
68       int main(int, char**)
69       {
70           MyBuffer buf;
71           [illegible] dummy;
72           tid1 = thread_create([illegible], &buf);
73           tid2 = [illegible] &buf)
74
75           // never reach this point
76           thread_join(tid1);
77           thread_join(tid2);
78           return [illegible]
79       }
80
81       void producer(void* buf)
82       {
83           MyBuffer* sharedbuf = reinterpret[illegible]<MyBuffer*>(buf);
84           for (;;) {
85               /* next line produces an item and puts it in nextProduced */
86               Item nextProduced = [illegible] of production();
87               sharedbuf->Enqueue[illegible]
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
```

2. This monitor is a model ... 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 ... As in the previous example, this code is expressed in pseudocode.

```
    // assume that these variables are initialized in a constructor
    state variables:
        AR = 0;  // # active readers
        AW = 0;  // # active writers
        WR = 0;  // # waiting readers
        WW = 0;  // # waiting writers

        Condition okToRead = NIL;
        Condition okToWrite = NIL;
        Mutex mutex = FREE

    Database::read() {
        startRead();  // first, check self into the system
        Access Data
        doneRead();   // check self out of system
    }

    Database::startRead() {
        acquire(&mutex);
        while((AW + WW) > 0) {
            WR++;
            wait(&okToRead, &mutex);
            WR--;
        }
        AR++;
        release(&mutex);
    }

   Database::doneRead() {
        acquire(&mutex);
        AR--;
        if (AR == 0 && WW > 0) { // if no other readers still
          signal(&okToWrite, &mutex);   // active, wake up writer
        }
        release(&mutex);
    }

    Database::write() { // symmetrical
        startWrite();  // check in
        Access Data
        doneWrite();   // check out
    }

    Database::startWrite() {
        acquire(&mutex);
        while ((AW + AR) > 0) { // check if safe to write.
                                // if any ...
            WW++;
            wait(&okToWrite, &mutex);
            WW--;
        }
        AW++;
        release(&mutex);
    }

    Database::doneWrite() {
        acquire(&mutex);
        AW--;
        if (WW > 0) {
            signal(&okToWrite, &mutex); // ... priority ... writers
        } else if (WR > 0) {
            broadcast(&okToRead, &mutex);
        }
        release(&mutex);
    }

    NOTE: what is the ... problem here?
```

3. Shared Locks

```
    struct sharedlock {
      int i;
      Mutex mutex
      Cond c;
    };

    void AcquireExclusive (sharedlock *sl) {
      acquire(&sl->mutex);
      while (sl->i)
        wait (&sl->c, &sl->mutex);
      }
      sl->i = -1;
      release(&sl->mutex);
    }

    void AcquireShared (sharedlock *sl) {
      acquire(&sl->mutex);
      while (sl->i < 0) {
        wait (&sl->c, &sl->mutex);
      }
      sl->i++;
      release(&sl->mutex);
    }

    void ReleaseShared (sharedlock *sl) {
      acquire(&sl->mutex);
      if (--sl->i == 0)
        signal(&sl->c, &sl->mutex);
      release(&sl->mutex);
    }

    void ReleaseExclusive (sharedlock *sl) {
      acquire(&sl->mutex);
      sl->i = 0;
      broadcast (&sl->c, &sl->mutex);
      release(&sl->mutex);
    }
```

QUESTIONS:

A. There is a starvation problem here. What is it? ... readers starve writers out ...

B. How could you use this shared lock ... to write a cleaner version of the ... code ... that the starvation properties would be different ...

Im[illegible]

1. Here is [illegible] spinlock implementation:

```
struct Spinlock {
  int locked;
[illegible]

[illegible]
  [illegible]
    [illegible] (lock->locked == 0) [illegible]
      lock->locked = 1
      break;
    }
  }
}

void release (Spinlock *lock) {
  lock->locked = 0;
}
```

[illegible]
CP[illegible] might [illegible]
[illegible] think the [illegible]
[illegible] mutual [illegible].

2. Correct spinlock implementation

Relies on atomic [illegible] instruction. For example, on the x86-64,
doing
"xchg [illegible] %rax"
does the [illegible]

(i) freeze [illegible] CPUs [illegible] activity for address addr
(ii) [illegible]
(iii) [illegible] %rax
(iv) [illegible]
(v) [illegible] freeze [illegible] activity

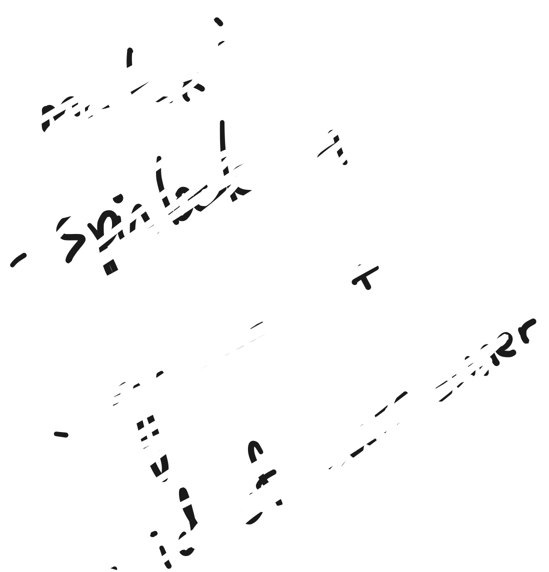
```
/* pseudocode
int xchg_val( [illegible]
    %rax [illegible]
    xchg [illegible], %rax
}

/* [illegible] of acquire */
void acquire (Spinlock *lock) {
  pushcli(); /* [illegible] this [illegible]
  while (1) {
    if (xchg_val(&lock->locked, 1) [illegible]
      break;
  }
}

void release(Spinlock *lock) {
  xchg_val(&lock->locked, 0);
  [illegible]
}

/* optimization [illegible] xchg_val() less frequently */
void acquire(Spinlock *lock) {
    pushcli();
    while (xchg_val(&lock->locked, 1) == 1) {
        while (lock->locked) ;
    }
}
```

The [illegible] is called [illegible] spinlock [illegible] spins. [illegible]
[illegible] is [illegible] spinlock [illegible]
[illegible]

[illegible]

[illegible] the [illegible] performance [illegible]
[illegible] this [illegible]
[illegible] (use [illegible]
[illegible] CPUs [illegible]
[illegible]

[illegible] very [illegible] application-level programming, spinlocks
will not be [illegible] (use mutexes instead). But you should
know what these are for technical literacy and to see where the
mutual exclusion is truly enforced on modern hardware.



```
} thread_t;

    thread_t *owner;

    // List of threads waiting on mutex

    // A lock protecting the internals of the mutex.
    Spinlock splock;    // as in item 1, above
}

void mutex_acquire(struct Mutex *m) {

        // ... if not, current thread gets mutex and returns
        if (...) {
            m->owner = id_of_this_thread;
            release(&m->splock);
        } else {
            // Add thread to waiters
            STAILQ_INSERT_TAIL(&m->waiters, id_of_this_thread, qlink);

            // Tell the scheduler to add current thread to the
            // list of blocked threads. The scheduler needs to be careful
            // when a corresponding sched_suspend call is executed to
            // make sure that it treats running threads correctly.
            sched_mark_blocked(&id_of_this_thread);

            // Unlock spinlock
            release(&m->splock);

            // Stop executing until woken.
            sched_...

            // ... we are guaranteed to hold the mutex ...
            // ... mutex_release ...
            // ... might actually have held the mutex ...

            // ... after the spinlock release ...
            // ... it just means that we are
            // ... before proceeding.
        }
}

void mutex_release(struct Mutex *m) {
        // Acquire the spinlock in order to make changes.
        acquire(&m->splock);

        // Assert that the current thread actually owns the mutex
        assert(m->owner == id_of_this_thread);

        // Check if anyone is waiting.
        ... = STAILQ_GET_HEAD(&m->waiters);

        // ... wake them up.

        ... STAILQ_REMOVE_HEAD(&m->waiters, qlink);
    }

    // Release the internal spinlock
    release(&m->splock);
```

```
1  CS 202, Spring 2023
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).
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
```



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

```
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
```



```
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.
```

```
CS 202, Spring 2023
Handout 5 (Class 6)

The previous handout demonstrated the use of mutexes and condition
variables. This handout demonstrates the use of monitors (which combine
mutexes and condition variables).

1. The bounded buffer as a monitor

    // This is pseudocode that is inspired by C++.
    // Don't take it literally.

    class MyBuffer {
      public:
        MyBuffer();
        ~MyBuffer();
        void Enqueue(Item);
        Item = Dequeue();
      private:
        int count;
        int in;
        int out;
        Item buffer[BUFFER_SIZE];
        Mutex* mutex;
        Cond* nonempty;
        Cond* nonfull;
    }

    void
    MyBuffer::MyBuffer()
    {
        in = out = count = 0;
        mutex = new Mutex;
        nonempty = new Cond;
        nonfull = new Cond;
    }
```

Hoare-style

Hansen-style

```
    void
    MyBuffer::Enqueue(Item item)
    {
        mutex.acquire();
        while (count == BUFFER_SIZE)
            cond_wait(&nonfull, &mutex);
```

release();

acquire();

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

    Item
    MyBuffer::Dequeue()
    {
        mutex.acquire();
        while (count == 0)
            cond_wait(&nonempty, &mutex);

        Item ret = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        --count;
        cond_signal(&nonfull, &mutex);
        mutex.release();
        return ret;
    }
```



```
    int main(int, char**)
    {
        MyBuffer buf;
        int dummy;
        tid1 = thread_create(producer, &buf);
        tid2 = thread_create(consumer, &buf);

        // never reach this point
        thread_join(tid1);
        thread_join(tid2);
        return -1;
    }

    void producer(void* buf)
    {
        MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
        for (;;) {
            /* next line produces an item and puts it in nextProduced */
            Item nextProduced = means_of_production();
            sharedbuf->Enqueue(nextProduced);
        }
    }

    void consumer(void* buf)
    {
        MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
        for (;;) {
            Item nextConsumed = sharedbuf->Dequeue();

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

    Key point: *Threads* (the producer and consumer) are separate from
    *shared object* (MyBuffer). The synchronization happens in the
    shared object.
```

```
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();   // check self out of system
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);   // active, wake up writer
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.)
```

```
1  Implementation of spinlocks and mutexes
2
3  1. Here is a BROKEN spinlock implementation:
4
5          struct Spinlock {
6            int locked;
7          }
8
9          void acquire(Spinlock *lock) {
10           while (1) {
11             if (lock->locked == 0) { // A
12               lock->locked = 1;      // B
13               break;
14             }
15           }
16         }
17
18         void release (Spinlock *lock) {
19           lock->locked = 0;
20         }
21
22         What's the problem? Two acquire()s on the same lock on different
23         CPUs might both execute line A, and then both execute B. Then
24         both will think they have acquired the lock. Both will proceed.
25         That doesn't provide mutual exclusion.
26
```



```
26
27 2. Correct spinlock implementation
28
29     Relies on atomic hardware instruction. For example, on the x86-64,
30         doing
31                 "xchg addr, %rax"
32         does the following:
33
34         (i)   freeze all CPUs' memory activity for address addr
35         (ii)  temp <-- *addr
36         (iii) *addr <-- %rax
37         (iv)  %rax <-- temp
38         (v)   un-freeze memory activity
39
40     /* pseudocode */
41     int xchg_val(addr, value) {
42         %rax = value;
43         xchg (*addr), %rax
44     }
45
46     /* bare-bones version of acquire */
47     void acquire (Spinlock *lock) {
48       pushcli();    /* what does this do? */
49       while (1) {
50         if (xchg_val(&lock->locked, 1) == 0)
51           break;
52       }
53     }
54
55     void release(Spinlock *lock){
56        xchg_val(&lock->locked, 0);
57        popcli();    /* what does this do? */
58     }
59
60
61     /* optimization in acquire; call xchg_val() less frequently */
62     void acquire(Spinlock* lock) {
63         pushcli();
64         while (xchg_val(&lock->locked, 1) == 1) {
65             while (lock->locked) ;
66         }
67     }
68
69     The above is called a *spinlock* because acquire() spins. The
70     bare-bones version is called a "test-and-set (TAS) spinlock"; the
71     other is called a "test-and-test-and-set spinlock".
72
73     The spinlock above is great for some things, not so great for
74     others. The main problem is that it *busy waits*: it spins,
75     chewing up CPU cycles. Sometimes this is what we want (e.g., if
76     the cost of going to sleep is greater than the cost of spinning
77     for a few cycles waiting for another thread or process to
78     relinquish the spinlock). But sometimes this is not at all what we
79     want (e.g., if the lock would be held for a while: in those
80     cases, the CPU waiting for the lock would waste cycles spinning
81     instead of running some other thread or process).
82
83     NOTE: the spinlocks presented here can introduce performance issues
84     when there is a lot of contention. (This happens even if the
85     programmer is using spinlocks correctly.) The performance issues
86     result from cross-talk among CPUs (which undermines caching and
87     generates traffic on the memory bus). If we have time later, we will
88     study a remediation of this issue (search the Web for "MCS locks").
89
90     ANOTHER NOTE: In everyday application-level programming, spinlocks
91     will not be something you use (use mutexes instead). But you should
92     know what these are for technical literacy, and to see where the
93     mutual exclusion is truly enforced on modern hardware.
94
```

```
95  3. Mutex implementation
96
97      The intent of a mutex is to avoid busy waiting: if the lock is not
98      available, the locking thread is put to sleep, and tracked by a
99      queue in the mutex. The next page has an implementation.
100
101
```



```c
1   #include <sys/queue.h>
2
3   typedef struct thread {
4       // ... Entries elided.
5       STAILQ_ENTRY(thread_t) qlink; // Tail queue entry.
6   } thread_t;
7
8   struct Mutex {
9       // Current owner, or 0 when mutex is not held.
10      thread_t *owner;
11
12      // List of threads waiting on mutex
13      STAILQ(thread_t) waiters;
14
15      // A lock protecting the internals of the mutex.
16      Spinlock splock;   // as in item 1, above
17  };
18
19  void mutex_acquire(struct Mutex *m) {
20
21      acquire(&m->splock);
22
23      // Check if the mutex is held; if not, current thread gets mutex and returns
24      if (m->owner == 0) {
25          m->owner = id_of_this_thread;
26          release(&m->splock);
27      } else {
28          // Add thread to waiters.
29          STAILQ_INSERT_TAIL(&m->waiters, id_of_this_thread, qlink);
30
31          // Tell the scheduler to add current thread to the list
32          // of blocked threads. The scheduler needs to be careful
33          // when a corresponding sched_wakeup call is executed to
34          // make sure that it treats running threads correctly.
35          sched_mark_blocked(&id_of_this_thread);
36
37          // Unlock spinlock.
38          release(&m->splock);
39
40          // Stop executing until woken.
41          sched_swtch();
42
43          // When we get to this line, we are guaranteed to hold the mutex. This
44          // is because we can get here only if context-switched-TO, which itself
45          // can happen only if this thread is removed from the waiting queue,
46          // marked "unblocked", and set to be the owner (in mutex_release()
47          // below). However, we might have held the mutex in lines 39-42
48          // (if we were context-switched out after the spinlock release(),
49          // followed by being run as a result of another thread's release of the
50          // mutex). But if that happens, it just means that we are
51          // context-switched out an "extra" time before proceeding.
52      }
53  }
54
55  void mutex_release(struct Mutex *m) {
56      // Acquire the spinlock in order to make changes.
57      acquire(&m->splock);
58
59      // Assert that the current thread actually owns the mutex
60      assert(m->owner == id_of_this_thread);
61
62      // Check if anyone is waiting.
63      m->owner = STAILQ_GET_HEAD(&m->waiters);
64
65      // If so, wake them up.
66      if (m->owner) {
67          sched_wakeone(&m->owner);
68          STAILQ_REMOVE_HEAD(&m->waiters, qlink);
69      }
70
71      // Release the internal spinlock
72      release(&m->splock);
73  }
```