

Aug 31, 2025 12:51

handout05.txt

Page 1/4

```

1 CS 202, Fall 2023
2 Handout 5 (Class 7)
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

```

Aug 31, 2025 12:51

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

```

Aug 31, 2025 12:51

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
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?

```

Aug 31, 2025 12:51

handout05.txt

Page 4/4

```

179 3. Shared locks
180
181 struct sharedlock {
182     int i;
183     Mutex mutex;
184     Cond c;
185 };
186
187 void AcquireExclusive (sharedlock *sl) {
188     acquire(&sl->mutex);
189     while (sl->i) {
190         wait (sl->c, &sl->mutex);
191     }
192     sl->i = -1;
193     release(&sl->mutex);
194 }
195
196 void AcquireShared (sharedlock *sl) {
197     acquire(&sl->mutex);
198     while (sl->i < 0) {
199         wait (sl->c, &sl->mutex);
200     }
201     sl->i++;
202     release(&sl->mutex);
203 }
204
205 void ReleaseShared (sharedlock *sl) {
206     acquire(&sl->mutex);
207     if (!--sl->i)
208         signal (&sl->c, &sl->mutex);
209     release(&sl->mutex);
210 }
211
212 void ReleaseExclusive (sharedlock *sl) {
213     acquire(&sl->mutex);
214     sl->i = 0;
215     broadcast (&sl->c, &sl->mutex);
216     release(&sl->mutex);
217 }
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.)

```

Aug 31, 2025 12:51

spinlock-mutex.txt

Page 1/3

```

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

```

Aug 31, 2025 12:51

spinlock-mutex.txt

Page 2/3

```

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

```

Aug 31, 2025 12:51

spinlock-mutex.txt

Page 3/3

86 3. Mutex implementation

87
88
89
90
91
92

The intent of a mutex is to avoid busy waiting: if the lock is not available, the locking thread is put to sleep, and tracked by a queue in the mutex. The next page has an implementation.

Aug 31, 2025 12:51

fair-mutex.c

Page 1/1

```

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_swch();
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 }

```