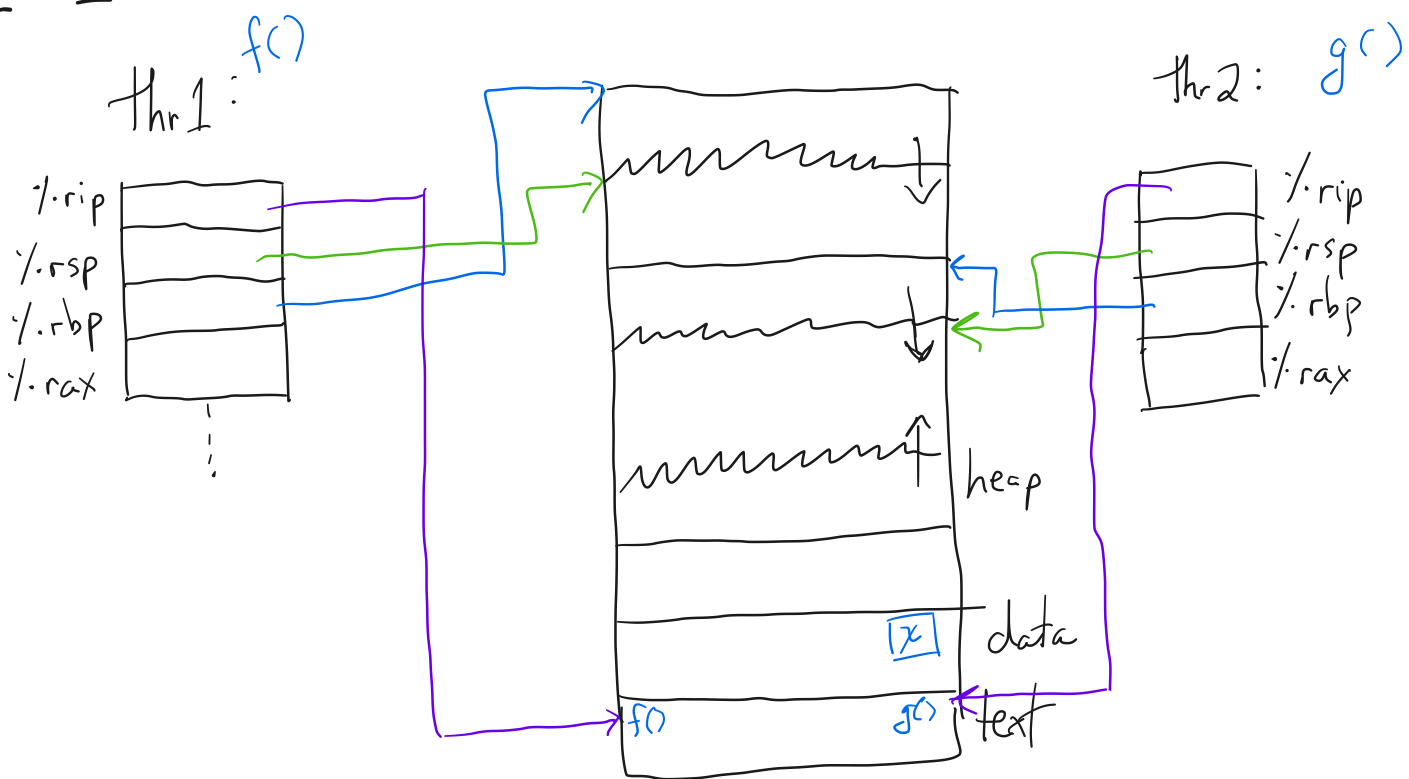


- ☑ 1. Last time
- ☑ 2. Intro to concurrency, continued
- ☑ 3. Managing concurrency
- ☑ 4. Mutexes
- ☑ 5. Condition variables
- ☑ 6. Semaphores

2. Intro to concurrency

- panels 1-3 on handout 03: all examples of "race conditions" (uncontrolled access to shared memory)
- hardware makes the problem harder (see panel 4)

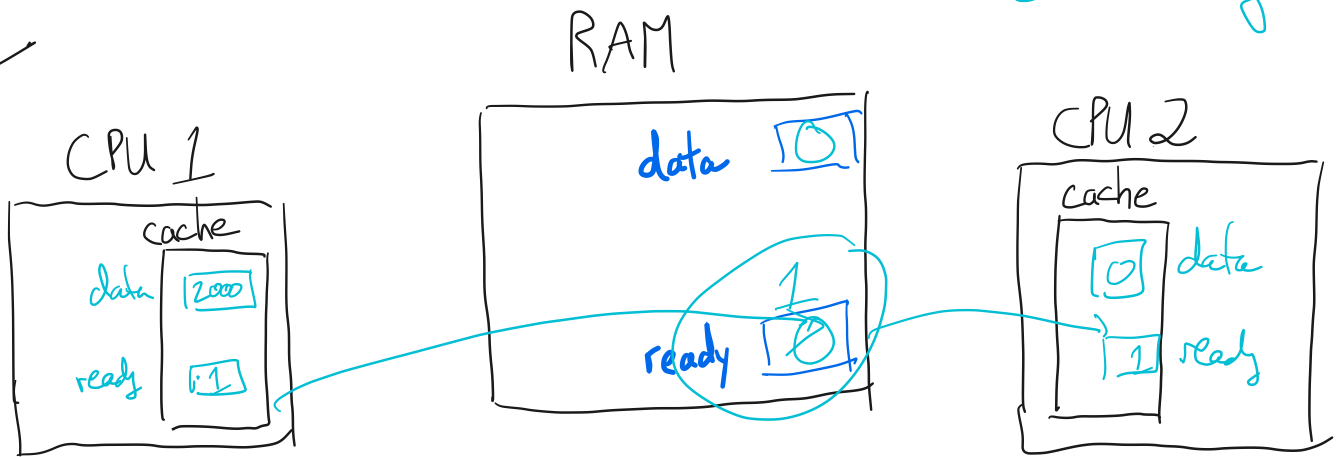


Threads share memory, but they have their own "execution context" (registers and stack).

To the programmer, it "feels like" multiple things are happening at once in the program.

memory consistency

Sequential consistency.



3. Managing concurrency

a. Critical sections: the concept: "protect from concurrent execution".

- i. mutual exclusion
- ii. progress
- iii. bounded waiting

b. Protecting critical sections

lock() / unlock()

enter() / leave()

acquire() / release()

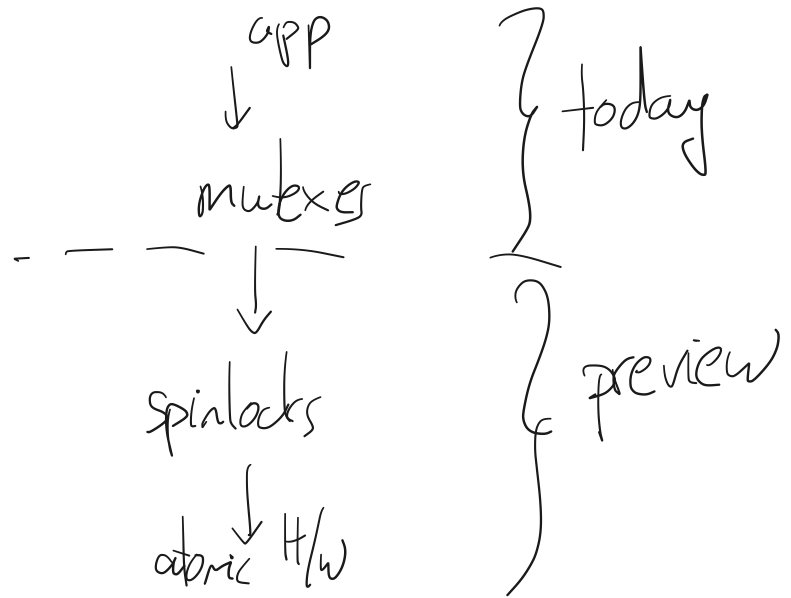
Implementing critical sections

c. Implementing critical sections

(i) single-CPU machine: `enter()` → disable interrupts

`leave()` → enable interrupts

4. Mutexes



5. Condition variables

`cond_int (Cond*, --);`

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

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

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

Feb 01, 23 0:13

handout03.txt

Page 1/4

```

1 CS 202, Spring 2023
2 Handout 3 (Class 4)
3
4 1. Example to illustrate interleavings: say that thread tid1 executes f()
5 and thread tid2 executes g(). (Here, we are using the term "thread"
6 abstractly. This example applies to any of the approaches that fall
7 under the word "thread".)

```

a. [this is pseudocode]

```

8
9     int x;
10
11     int main(int argc, char** argv) {
12
13         tid tid1 = thread_create(f, NULL);
14         tid tid2 = thread_create(g, NULL);
15
16         thread_join(tid1);
17         thread_join(tid2);
18
19         printf("%d\n", x);
20     }
21
22
23     void f()
24     {
25         x = 1;
26         thread_exit();
27     }
28
29     void g()
30     {
31         x = 2;
32         thread_exit();
33     }
34
35
36

```

What are possible values of x after A has executed f() and B has executed g()? In other words, what are possible outputs of the program above?

b. Same question as above, but f() and g() are now defined as follows:

```

43     int y = 12;
44
45     f() { x = y + 1; }
46     g() { y = y * 2; }
47
48

```

What are the possible values of x?

c. Same question as above, but f() and g() are now defined as follows:

```

55     int x = 0;
56     f() { x = x + 1; }
57     g() { x = x + 2; }
58
59

```

What are the possible values of x?

63

Feb 01, 23 0:13

handout03.txt

Page 2/4

64 2. Linked list example

```

65
66     struct List_elem {
67         int data;
68         struct List_elem* next;
69     };
70
71     List_elem* head = 0;
72
73     insert(int data) {
74         List_elem* l = new List_elem;
75         l->data = data;
76         l->next = head;
77         head = l;
78     }
79

```

80 What happens if two threads execute insert() at once and we get the following interleaving?

```

81
82
83     thread 1: l->next = head
84     thread 2: l->next = head
85     thread 2: head = l;
86     thread 1: head = l;
87

```

Feb 01, 23 0:13

handout03.txt

Page 3/4

```

88 3. Producer/consumer example:
89
90  /*
91  "buffer" stores BUFFER_SIZE items
92  "count" is number of used slots. a variable that lives in memory
93  "out" is next empty buffer slot to fill (if any)
94  "in" is oldest filled slot to consume (if any)
95  */
96
97  void producer (void *ignored) {
98      for (;;) {
99          /* next line produces an item and puts it in nextProduced */
100         nextProduced = means_of_production();
101         while (count == BUFFER_SIZE)
102             ; // do nothing
103         buffer [in] = nextProduced;
104         in = (in + 1) % BUFFER_SIZE;
105         count++;
106     }
107 }
108
109 void consumer (void *ignored) {
110     for (;;) {
111         while (count == 0)
112             ; // do nothing
113         nextConsumed = buffer[out];
114         out = (out + 1) % BUFFER_SIZE;
115         count--;
116         /* next line abstractly consumes the item */
117         consume_item(nextConsumed);
118     }
119 }
120
121 /*
122 what count++ probably compiles to:
123 reg1 <-- count      # load
124 reg1 <-- reg1 + 1   # increment register
125 count <-- reg1     # store
126
127 what count-- could compile to:
128 reg2 <-- count     # load
129 reg2 <-- reg2 - 1  # decrement register
130 count <-- reg2    # store
131 */
132
133 What happens if we get the following interleaving?
134
135 reg1 <-- count
136 reg1 <-- reg1 + 1
137 reg2 <-- count
138 reg2 <-- reg2 - 1
139 count <-- reg1
140 count <-- reg2
141

```

Feb 01, 23 0:13

handout03.txt

Page 4/4

```

142
143 4. Some other examples. What is the point of these?
144
145 [From S.V. Adve and K. Gharachorloo, IEEE Computer, December 1996,
146 66-76. http://rsim.cs.uiuc.edu/~sadve/Publications/computer96.pdf]
147
148 a. Can both "critical sections" run?
149
150     int flag1 = 0, flag2 = 0;
151
152     int main () {
153         tid id = thread_create (p1, NULL);
154         p2 (); thread_join (id);
155     }
156
157     void p1 (void *ignored) {
158         flag1 = 1;
159         if (!flag2) {
160             critical_section_1 ();
161         }
162     }
163
164     void p2 (void *ignored) {
165         flag2 = 1;
166         if (!flag1) {
167             critical_section_2 ();
168         }
169     }
170
171 b. Can use() be called with value 0, if p2 and p1 run concurrently?
172
173     int data = 0, ready = 0;
174
175     void p1 () {
176         data = 2000;
177         ready = 1;
178     }
179     int p2 () {
180         while (!ready) {}
181         use(data);
182     }
183
184 c. Can use() be called with value 0?
185
186     int a = 0, b = 0;
187
188     void p1 (void *ignored) { a = 1; }
189
190     void p2 (void *ignored) {
191         if (a == 1)
192             b = 1;
193     }
194
195     void p3 (void *ignored) {
196         if (b == 1)
197             use (a);
198     }
199

```

Feb 05, 23 23:11

handout04.txt

Page 1/4

```

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

```

Feb 05, 23 23:11

handout04.txt

Page 2/4

```

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

```

invariant

Feb 05, 23 23:11

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     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, 23 23:11

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.

```