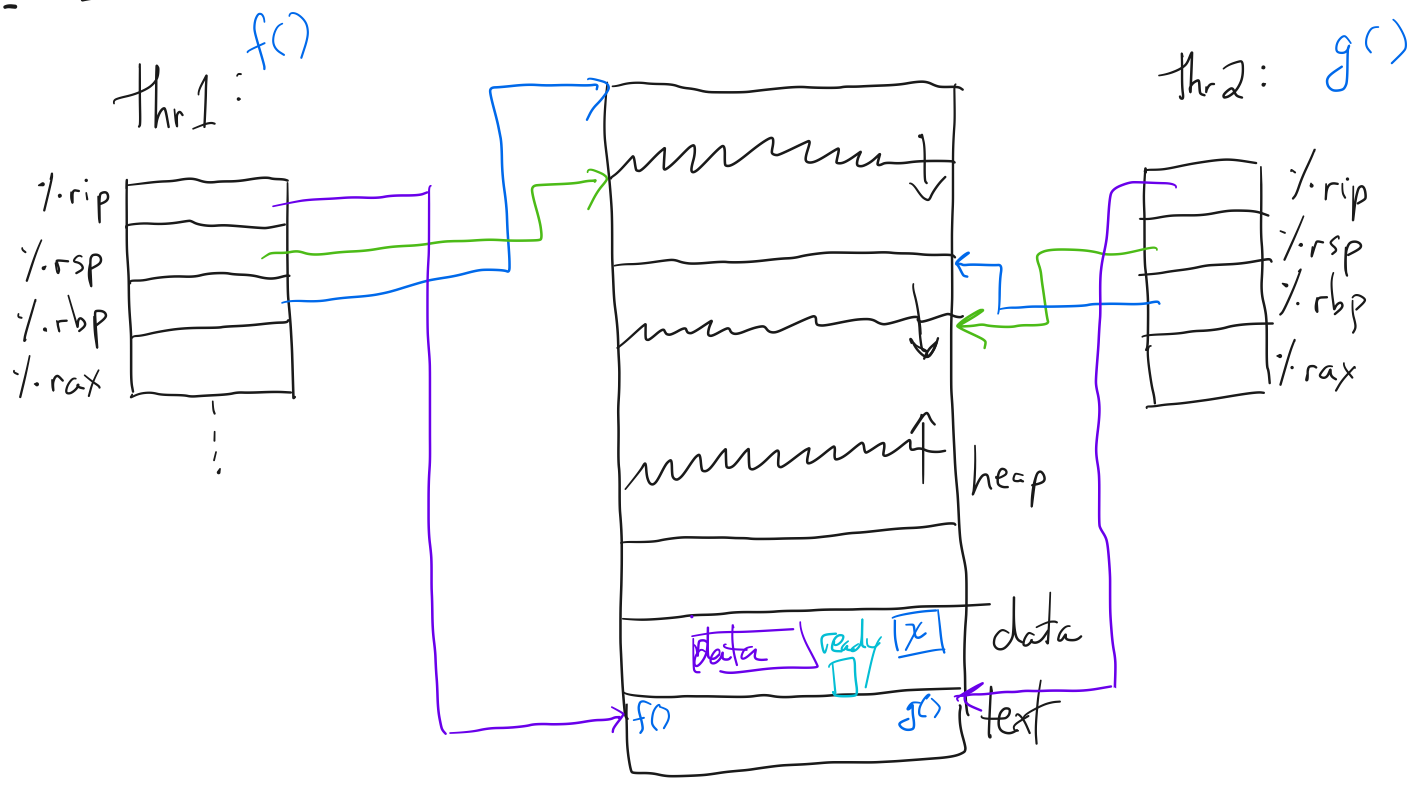
- [ ] 1. Last time
- [ ] 2. Intro to concurrency, continued
- [ ] 3. Managing concurrency
- [ ] 4. Mutexes
- [ ] 5. Condition variables
- [ ] 6. Semaphores

---

2. Intro to concurrency

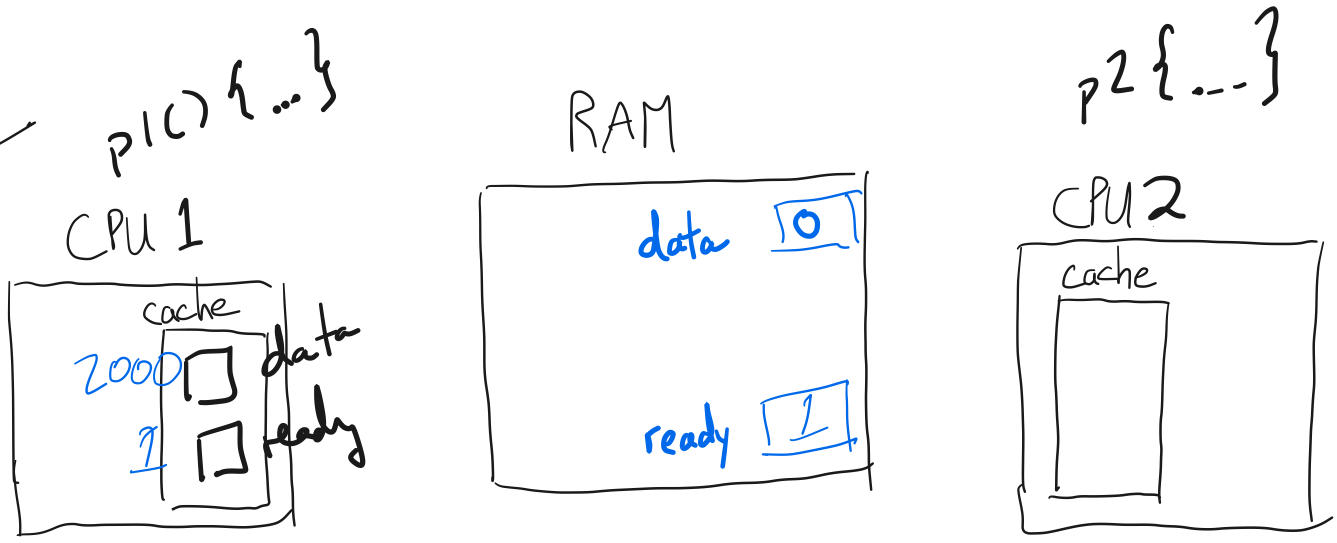
- panels 1-3 on handout03: all examples of "race conditions" (uncontrolled access to shared memory)
- hardware makes the problem harder (see panel 4)



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



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()

Two-thread critical sections

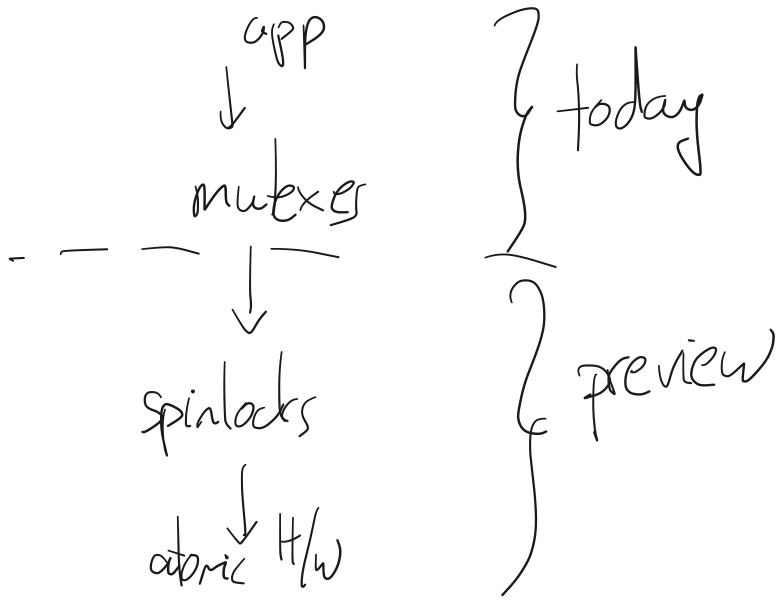
c. Implementing critical sections

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

leave() → enable interrupts

---

4. Mutexes



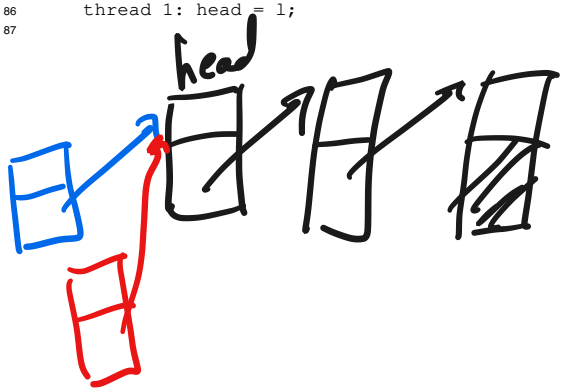
5. Condition variables

```
cond_init (Cond*, ...);

cond_wait(Mutex* m, Cond*);

cond_signal (Mutex* m, Cond*);

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

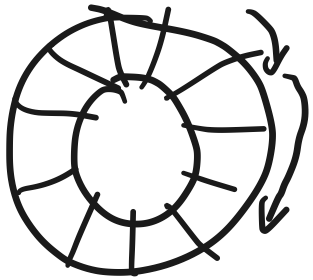
```
CS 202, Spring 2022
Handout 3 (Class 4)

1. Example to illustrate interleavings: say that thread A executes f()
and thread B executes g(). (Here, we are using the term "thread"
abstractly. This example applies to any of the approaches that fall
under the word "thread".)

    a. [this is pseudocode]

        int x;

        int main(int argc, char** argv) {

            tid tid1 = thread_create(f, NULL);
            tid tid2 = thread_create(g, NULL);

            thread_join(tid1);
            thread_join(tid2);

            printf("%d\n", x);
        }

        void f()
        {
            x = 1;
            thread_exit();
        }

        void g()
        {
            x = 2;
            thread_exit();
        }


        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:

        int y = 12;

        f() { x = y + 1; }
        g() { y = y * 2; }

        What are the possible values of x?



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

        int x = 0;
        f() { x = x + 1; }
        g() { x = x + 2; }

        What are the possible values of x?

```



```
2. Linked list example

    struct List_elem {
        int data;
        struct List_elem* next;
    };

    List_elem* head = 0;

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

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

    thread 1: l->next = head
    thread 2: l->next = head
    thread 2: head = l;
    thread 1: head = l;

```



```
3. Producer/consumer example:

    /*
    "buffer" stores BUFFER_SIZE items
    "count" is number of used slots. a variable that lives in memory
    "out" is next empty buffer slot to fill (if any)
    "in" is oldest filled slot to consume (if any)
    */

     void producer (void *ignored) {
         for (;;) {
             /* next line produces an item and puts it in nextProduced */
             nextProduced = means_of_production();
             while (count == BUFFER_SIZE)
                 ; // do nothing
             buffer [in] = nextProduced;
             in = (in + 1) % BUFFER_SIZE;
             count++;
         }
     }

     void consumer (void *ignored) {
         for (;;) {
             while (count <= 0)
                 ; // do nothing
             nextConsumed = buffer[out];
             out = (out + 1) % BUFFER_SIZE;
             count--;
             /* next line abstractly consumes the item */
             consume_item(nextConsumed);
         }
     }

    /*
       what count++ probably compiles to:
        reg1 <-- count      # load
        reg1 <-- reg1 + 1   # increment register
        count <-- reg1      # store

       what count-- could compile to:
        reg2 <-- count      # load
        reg2 <-- reg2 - 1   # decrement register
        count <-- reg2      # store
    */

    What happens if we get the following interleaving?

        reg1 <-- count
        reg1 <-- reg1 + 1
        reg2 <-- count
        reg2 <-- reg2 - 1
        count <-- reg1
        count <-- reg2
```




```
4. Some other examples. What is the point of these?

     [From S.V. Adve and K. Gharachorloo, IEEE Computer, December 1996,
     66-76. http://rsim.cs.uiuc.edu/~sadve/Publications/computer96.pdf]

     a. Can both "critical sections" run?

         int flag1 = 0, flag2 = 0;

         int main () {
             tid id = thread_create (p1, NULL);
             p2 (); thread_join (id);
         }

         void p1 (void *ignored) {
             flag1 = 1;
             if (!flag2) {
                 critical_section_1 ();
             }
         }

         void p2 (void *ignored) {
             flag2 = 1;
             if (!flag1) {
                 critical_section_2 ();
             }
         }

     b. Can use() be called with value 0, if p2 and p1 run concurrently?

         int data = 0, ready = 0;

         void p1 () {
             data = 2000;
             ready = 1;
         }
         int p2 () {
             while (!ready) {}
             use(data);
         }

     c. Can use() be called with value 0?

         int a = 0, b = 0;

         void p1 (void *ignored) { a = 1; }

         void p2 (void *ignored) {
           if (a == 1)
             b = 1;
         }

         void p3 (void *ignored) {
           if (b == 1)
             use (a);
         }
```

```
CS 202, Spring 2022
Handout 4 (Class 5)

The handout from the last class 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);
          }
      }
```

```
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.
```