Synchronization barrier problem

Glenn Bruns
CSUMB

Puzzle: rendezvous with n threads?

thread

```
1   rendezvous
2   critical point
```

Exercise: write code that will guarantee that no thread executes critical point until all threads have executed rendezvous.

This is called a **barrier**.

Assume each thread can access a constant ***n***, which is the number of threads.

Hint: you can introduce variables, and can use if statements

Hint: access to shared variables must be mutually exclusive

Hint 1

hint:

```
1   n = the number of threads
2   count = 0
3   mutex = Semaphore(1)
4   barrier = Semaphore(0)
```

count – how many threads are at the barrier

mutex – provides mutually exclusive access to count

barrier – locked until all threads arrive

thread

```
1   rendezvous
2   critical point
```

Hint 2

hint:

```
1    rendezvous
2
3    mutex.wait()
4       count = count + 1
5    mutex.signal()
6
7    if count == n: barrier.signal()
8
9    barrier.wait()
10
11   critical point
```

this *almost* works

what is wrong?

how to fix it?

Solution

hint:

```
1    rendezvous
2
3    mutex.wait()
4       count = count + 1
5    mutex.signal()
6
7    if count == n: barrier.signal()
8
9    barrier.wait()
10   barrier.signal()
11
12   critical point
```

As each thread passes through the barrier, it signals that another thread can pass.

A wait followed by a signal is called a turnstile.

Is this a solution?

hint:

```
1    rendezvous
2
3    mutex.wait()
4       count = count + 1
5    mutex.signal()
6
7    if count < n: barrier.wait()
8
9    barrier.signal()
11
12   critical point
```

As each thread passes through the barrier, it signals that another thread can pass.

A wait followed by a signal is called a turnstile.

Synchronization barrier lab

```
// actor: prints something, waits for everyone, prints something else
void *actor(void *arg) {
  BARR *barr = (BARR *)arg;
  printf("before barrier\n");
  check_in(barr);
  printf("after barrier\n");
  pthread_exit(NULL);
}

// test the barrier
int main(int argc, char *argv[]) {
  int n = 10;
  pthread_t actors[n];

  BARR *barr = barr_create(n);

  // create some actors
  int i;
  for (i = 0; i < n; i++) {
    pthread_create(&actors[i], NULL, actor, (void *)barr);
  }

  pthread_exit(NULL);
}
```

Barrier object

```
typedef struct {
  // state variables
  // YOUR CODE HERE

  // sync variables
  // YOUR CODE HERE
} BARR;

BARR *barr_create(int num_t) {
  BARR *barr;
  barr = malloc(sizeof(BARR));
  assert(barr != NULL);

  // YOUR CODE HERE

  return(barr);
}

void check_in(BARR *barr) {

  // YOUR CODE HERE

}
```

The barrier object has only one operation: check_in

When a thread calls check_in, the check_in operation won't return until num_t threads have called it.

You have to fill in almost every detail of the design of the barricade object.

Use the Anderson/Dahlin method

- start with normal class design
  - identify state variables
  - implement methods
- add a lock to the class; enclose method bodies with lock/unlock calls
- add 0 or more condition variables to the class
- add wait calls within loops
- add signal and broadcast calls

no semaphores!

Using condition variables: wait

Using wait:

- think of the condition on shared variables that you're waiting for
- loop while this condition not true
- wait statement within loop

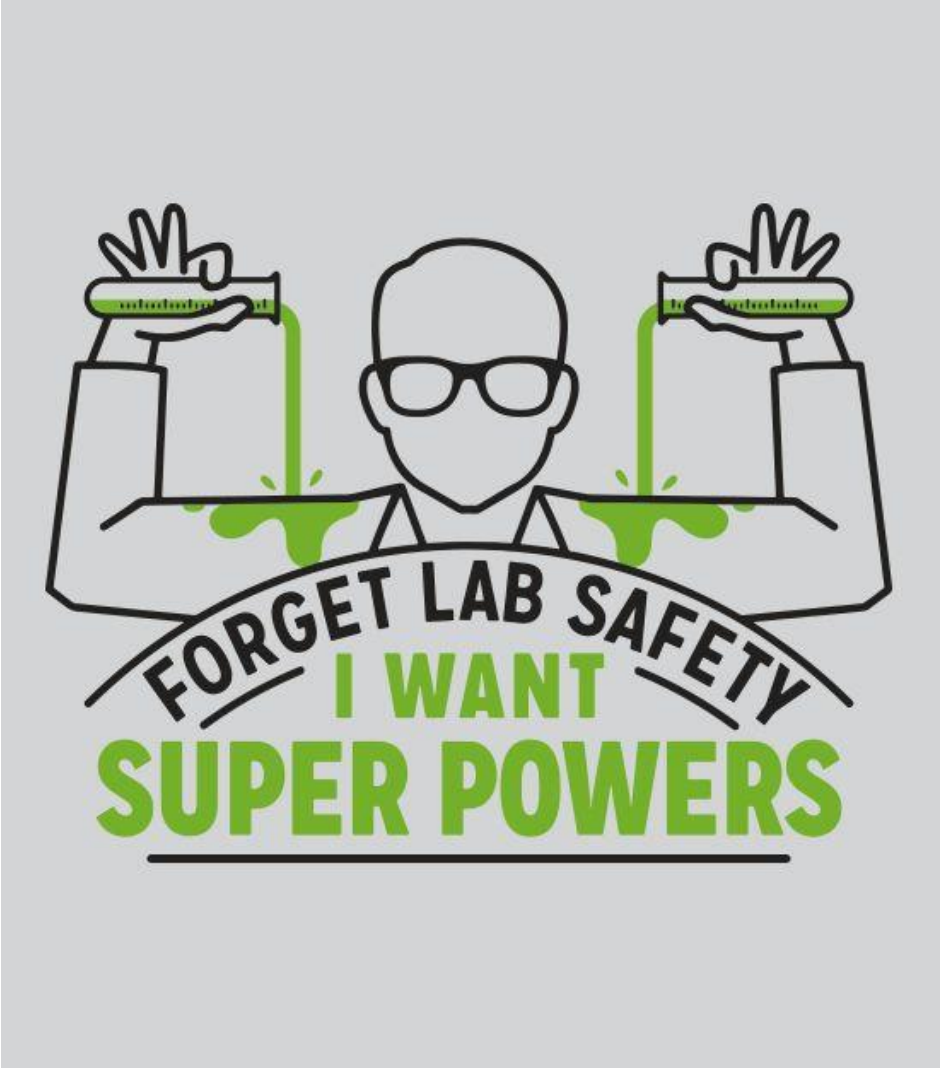
```
while (buffer is full) {
   pthread_cond_wait(&sbuf->write_go, &sbuf->lock);
}
```

Using condition variables: signal

Using wait:

- think of the threads that are interested in the way you've changed the state
- it's okay to "over-signal" Why?

Time for lab



Synchronization barrier shared state

Question 1:

What condition do we need to wait on?

Question 2:

When does a signal need to happen?

If you finish early

Create a reusable barrier.

- after all the threads have passed the barrier, the turnstile is locked again
- modify the test code so that the barrier is used repeatedly
- see Downey's Little Book of Semaphores if you need details

http://greenteapress.com/wp/semaphores/

If you still have time, look for further problems in the Little Book of Semaphores

- but solve them with locks and condition vars