Problem Solving Session

Monitors

Prepared by Dhruv Singhal

Bounded Buffer

The *bounded buffer* essentially implements a *blocking queue*. There is a shared class **`BoundedBuffer`** with a buffer of size **`N`**, a “head” index where new entries are inserted, and a “tail” index where new entries are removed. Initially, head and tail are 0. You are to write two methods:

- **`produce(item)`** adds a new item to the bounded buffer, and should block if the buffer is full.
- **`consume()`** removes and returns an item, and should block if the buffer is empty.

Use locks (mutexes) and Mesa-style condition variables. No busy waiting: when a thread cannot continue, it should block.

Implement the queue as a Python list of a fixed size.

Solution

```
class BoundedBuffer:
  def __init__(self, N):
    self.N = N
    self.buffer = [None] * N
    self.head = 0
    self.tail = 0
    self.lock = Lock()
    self.empty = Condition(self.lock)
    self.full = Condition(self.lock)
    self.n_entries = 0

  def produce(self, item):
    with self.lock:
      while self.n_entries == self.N:
        self.full.wait()
      self.buf[self.head] = item
      self.head = (self.head + 1) % self.N
      self.n_entries += 1
      self.empty.notify()
```

```
def consume(self):
  with self.lock:
    while self.n_entries == 0:
      self.empty.wait()
    item = self.buf[self.tail]
    self.tail = (self.tail + 1) % self.N
    self.n_entries -= 1
    self.full.notify()
    return item
```

Pool Hall

You are to simulate a pool hall with **`N`** pool tables numbered **`0`** to **`N - 1`**. Players (*threads*) that arrive at the pool hall are looking for tables that have one (preferred) or no players at them. Once there are two players at a table, they play a game of pool and then they leave (separately). When both players have left the pool table, the table becomes available for new players.

Use Mesa locks and condition variables. No busy waiting!

Solution

```
class PoolHall:
  def __init__(self, N):
    self.N = N
    self.lock = Lock()
    self.n_at_table = [0] * N
    self.n_left = [0] * N
    self.table_cond = [Condition(self.lock) \
      for _ in range(N)]
    self.avail_cond = Condition(self.lock)

  def table_avail(self):
    for tno in range(N):
      if self.n_at_table[tno] == 1:
        return tno
    for tno in range(N):
      if self.n_at_table[tno] == 0:
        return tno
    return -1
```

```
def player_enters(self):
  with self.lock:
    while self.table_avail() < 0:
      self.avail_cond.wait()
  tno = self.table_avail()
  self.n_at_table[tno] += 1
  if self.n_at_table[tno] != 2:
    self.table_cond[tno].wait()
  else:
    self.table_cond[tno].notify()
    self.n_left[tno] = 0
  return tno

def player_exits(self, tno):
  with self.lock:
    self.n_left[tno] += 1
    if self.n_left[tno] == 2:
      self.n_at_table[tnp] = 0
      self.avail_cond.notifyAll()
```