Central Practice Problems

- https://drive.google.com/drive/folders/1yxd8Fumkz0NtQ1SnErjt7AiIr_GmM-Fj?usp=sharing

Previous Slides

- Refer to Midterm Practice Problems Sheet
- https://docs.google.com/document/d/1J948j8dKgxcbPEaNTH8pyPXce7kej7Sa3T0_CrKR2Wk/edit?usp=drive_link

Slide 4 - Synchronization (After mid topics)

- Identify deadlock situations (i.e. identify the necessary conditions for deadlock) from code
- Fix deadlock situations
- Theory questions

Examples

1. Identify the error in the code

```
1   void camper(int id) {
2         do {
3               wait(pole[id % POLES]);
4               wait(pole[(id + 1) % POLES]);
5
6               /* Set up tent */
7
8               signal(pole[id % POLES]);
9               signal(pole[(id + 1) % POLES]);
10        } while (true);
11  }
```

Answer: The error in the code is that it can lead to a deadlock. If all campers simultaneously acquire one pole (e.g., pole[id % POLES]) and wait for the second pole (pole[(id + 1) % POLES]), which is held by another camper, a circular wait occurs, causing them to wait indefinitely since no one can proceed.

1. Mutual Exclusion: Satisfied. Each pole can be held by only one camper at a time, as implied by the wait() and signal() operations, which suggest exclusive access.
2. Hold and Wait: Satisfied. A camper holds one pole (after the first wait()) while waiting for the second pole (via the second wait()).
3. No Preemption: Satisfied. There is no mechanism in the code to force a camper to release a pole once acquired; they only release it after using both poles.
4. Circular Wait: Satisfied. If each camper takes one pole and waits for the next, a circular chain forms (e.g., Camper 1 waits for Camper 2's pole, Camper 2 waits for Camper 3's pole, and so on, looping back), causing all to wait indefinitely.

```
1  #define POLES 2
2  int pole[POLES];
3
4  void camper(int id) {
5        do {
6        if (id % 2 == 0) {
7              wait(pole[id % POLES]);
8              wait(pole[(id + 1) % POLES]);
9        } else {
10             wait(pole[(id + 1) % POLES]);
11             wait(pole[id % POLES]);
12       }
13       /* Set up tent */
14       signal(pole[id % POLES]);
15       signal(pole[(id + 1) % POLES]);
16       } while (true);
17 }
```

2. Identify the error in the code

```
1  void baker() {
2      while (true) {
3          if (count < SHELF_SIZE) {
4              shelf[count] = 1;
5              count++;
6          } else {
7              wait();
8          }
9      }
```

```
10  }
11
12  void cashier() {
13      while (true) {
14          if (count > 0) {
15              shelf[count - 1] = 0;
16              count--;
17              wakeup(baker);
18          } else {
19              wait();
20          }
21      }
22  }
```

Answer: The issue in the code is that it can lead to a deadlock. If the cashier checks the shelf and finds it empty (count == 0) but is interrupted before waiting, and meanwhile the baker fills the shelf and calls wakeup() (which has no effect since cashier isn't waiting yet), then when the cashier resumes and waits, and the baker later waits on a full shelf, both end up waiting indefinitely.

1. Mutual Exclusion: Satisfied. Access to the shelf slots is exclusive, as only one process (baker or cashier) can modify the count and shelf at a time, implied by the sequential checks and updates.
2. Hold and Wait: Satisfied. The baker holds shelf space (by incrementing count) while waiting to add more if the shelf is full, and the cashier waits to remove items while not holding any specific slot but still in a blocking state.
3. No Preemption: Satisfied. There is no mechanism to force the baker or cashier to release control of the shelf or wake up prematurely; they only proceed based on their own conditions.
4. Circular Wait: Satisfied. A circular dependency forms due to timing; the baker waits for the cashier to empty the shelf, and the cashier waits for the baker to fill it, but a lost wakeup signal causes neither to proceed, effectively stalling both.

```
1  void baker() {
2        while (true) {
3        wait(&empty_slots);
4        wait(&mutex);
```

```
5       shelf[count] = 1;
6       count++;
7       signal(&mutex);
8       signal(&filled_slots);
9       }
10  }
11
12  void cashier() {
13      while (true) {
14      wait(&filled_slots);
15      wait(&mutex);
16      shelf[count - 1] = 0;
17      count--;
18      signal(&mutex);
19      signal(&empty_slots);
20      }
21  }
```

Slide 5 - File Systems

- File Systems Practice Problems
- https://docs.google.com/document/d/1-avPExhjMiivzmgAtUwJgsoasK-O7Lkz4HPLo79aTt8/edit?usp=drive_link

Slide 6 - Memory Management

- Memory Management Practice Problems (I will add multi-level paging problems asap)
- https://docs.google.com/document/d/1h3ApDwQDJFROPq7OV4h0BN30I_3cXuQDI9KW7E4xVTE/edit?usp=drive_link