Solving Deadlock
On Thursday, we explored the four necessary conditions for deadlock. In the context of the dining philosophers problem, how do we remove each of the four?

1. Mutual Exclusion

2. Circular Wait

3. Hold and Wait

4. No Preemption

Deadlock Solution Considerations
1. Fairness:

2. Livelock:

```c
#define THINGS_MAX 10
#define THREAD_CT 5

int things[THINGS_MAX];
int things_ct = 0;

int main() {
    int i;
    // Create `thread_ct` threads of each producer and consumer:
    pthread_t tid_consumer[THREAD_CT];
    pthread_t tid_producer[THREAD_CT];
    for (i = 0; i < THREAD_CT; i++) {
        pthread_create(&tid_consumer[i], NULL, producer, NULL);
        pthread_create(&tid_producer[i], NULL, consumer, NULL);
    }
    // Join threads:
    for (i = 0; i < THREAD_CT; i++) {
        pthread_join(tid_consumer[i], NULL);
        pthread_join(tid_producer[i], NULL);
    }
}
```
Synchronization Primitives
In programming, a key synchronization primitive has evolved to become common features of many programming languages.

Primitive: _________________________
- Allow asynchronous execution until a ______________.
- In JavaScript and Python, this is _________________.

Multi-Threaded Uses
There are several different reasons you will come across the use of multi-threaded applications:

(1): Concurrent Compute
(2): User Interactivity
(3): Responsiveness in Requests
(4): NOT for Security
(5): NOT for Isolation

General Pattern:
- An “async” call returns a ______________, not a return value.
- The “await” call _______ and then ____________________.