Synchronization, Dining Philosophers, and Deadlock

CS 240 - The University of Illinois

Wade Fagen-Ulmschneider

September 23, 2021

Synchronization

Synchronization Technique #1

`pthread_mutex_init`: Creates a new lock in the “unlocked” state.

`pthread_mutex_lock(pthread_mutex_t *mutex)`:

- When `mutex` is unlocked, change the lock to the “locked” state and advance to the next line of code.
- When `mutex` is locked, this function **blocks** execution until the lock can be acquired.

`pthread_mutex_unlock`: Moves the lock to the “unlocked” state.

`pthread_mutex_destory`: Destroys the lock; frees memory.

09/count-with-lock.c

```c
pthread_mutex_t lock;
int ct = 0;

void *thread_start(void *ptr) {
  int countTo = *((int *)ptr);

  int i;
  for (i = 0; i < countTo; i++) {
    pthread_mutex_lock(&lock);
    ct = ct + 1;
    pthread_mutex_unlock(&lock);
  }

  return NULL;
}
```

Synchronization Technique #2

pthread_cond_init: Create a new conditional variable.

pthread_cond_wait(**pthread_cond_t *cond, pthread_mutex_t *mutex**): Performs two different synchronization actions:

-
-

pthread_cond_signal(**pthread_cond_t *cond**): Unblocks “at least one thread” that is blocked on `**cond**` (if any threads are blocked; otherwise an effective “NO OP”).

pthread_cond_broadcast(**pthread_cond_t *cond**): Unblocks **ALL** threads blocked on `**cond**`.

pthread_mutex_destory: Destroys the lock; frees memory.

09/count-with-lock.c

```c
int things[THINGS_MAX];
int things_ct = 0;

void *producer(void *vptr) {
  while (1) {
    pthread_mutex_lock(&lock);

    // Cannot produce until there's space:
    while (things_ct >= THINGS_MAX) {
      pthread_cond_wait(&cond, &lock);
    }

    // Produce a thing:
    things[things_ct] = rand();
    printf("Produced [%d]: %d\n", things_ct, things[things_ct]);
    things_ct++;

    // Signal any waiting consumers:
    pthread_cond_broadcast(&cond);

    pthread_mutex_unlock(&lock);
  }
}
```

Synchronization Technique #3

`sem_init`: Creates a new semaphore with a specified “value”.

`sem_wait`: When the value is greater than zero, decreases the value and continues. Otherwise, **blocks** until the value is non-zero.

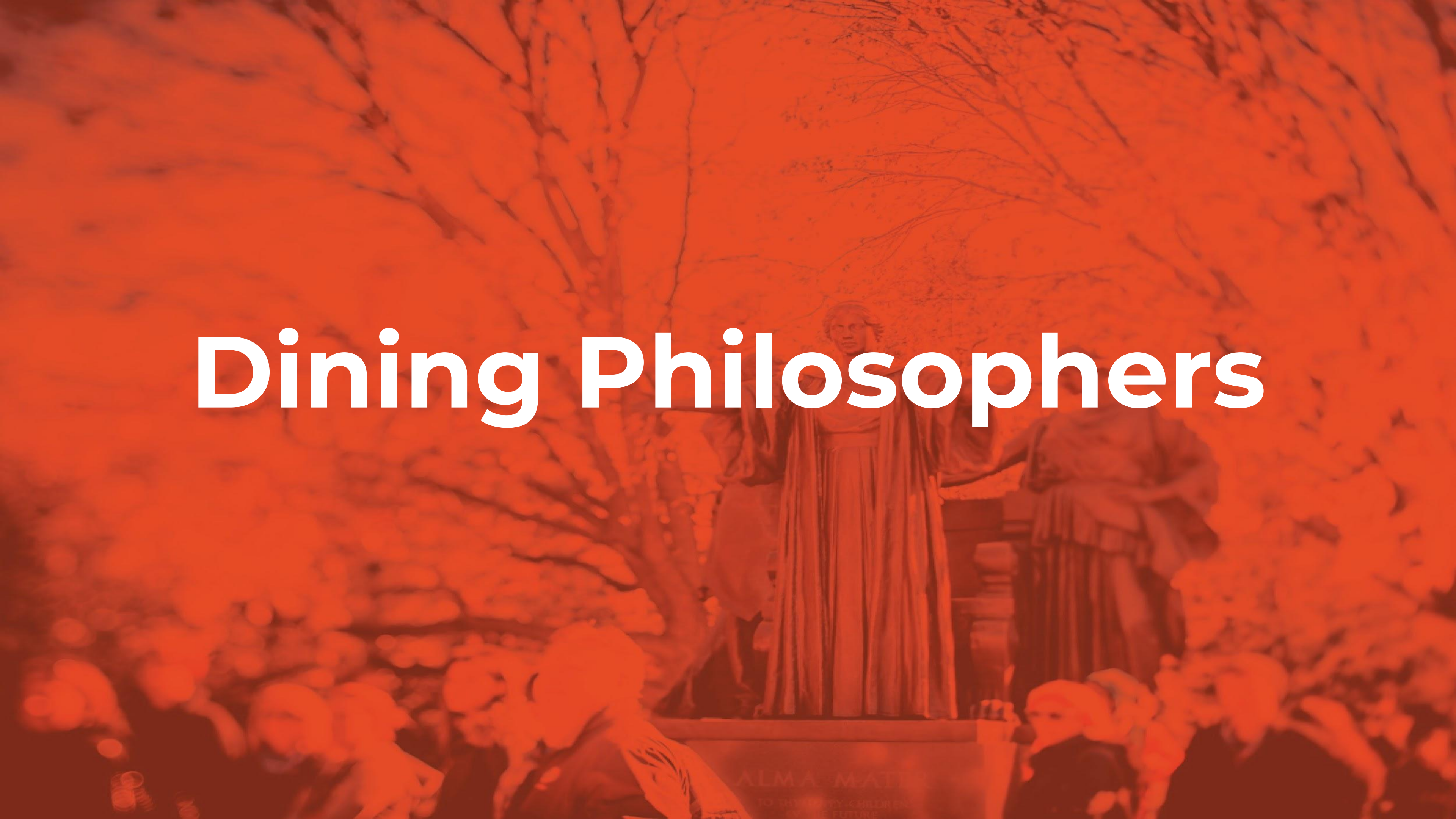
`sem_post`: Increments the value by one.

`sem_destroy`: Destroys the semaphore; frees memory.

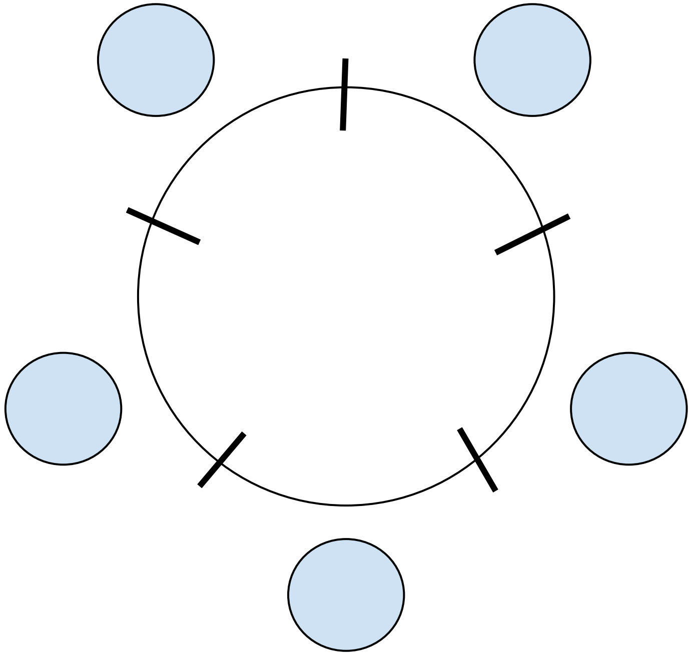
Critical Sections

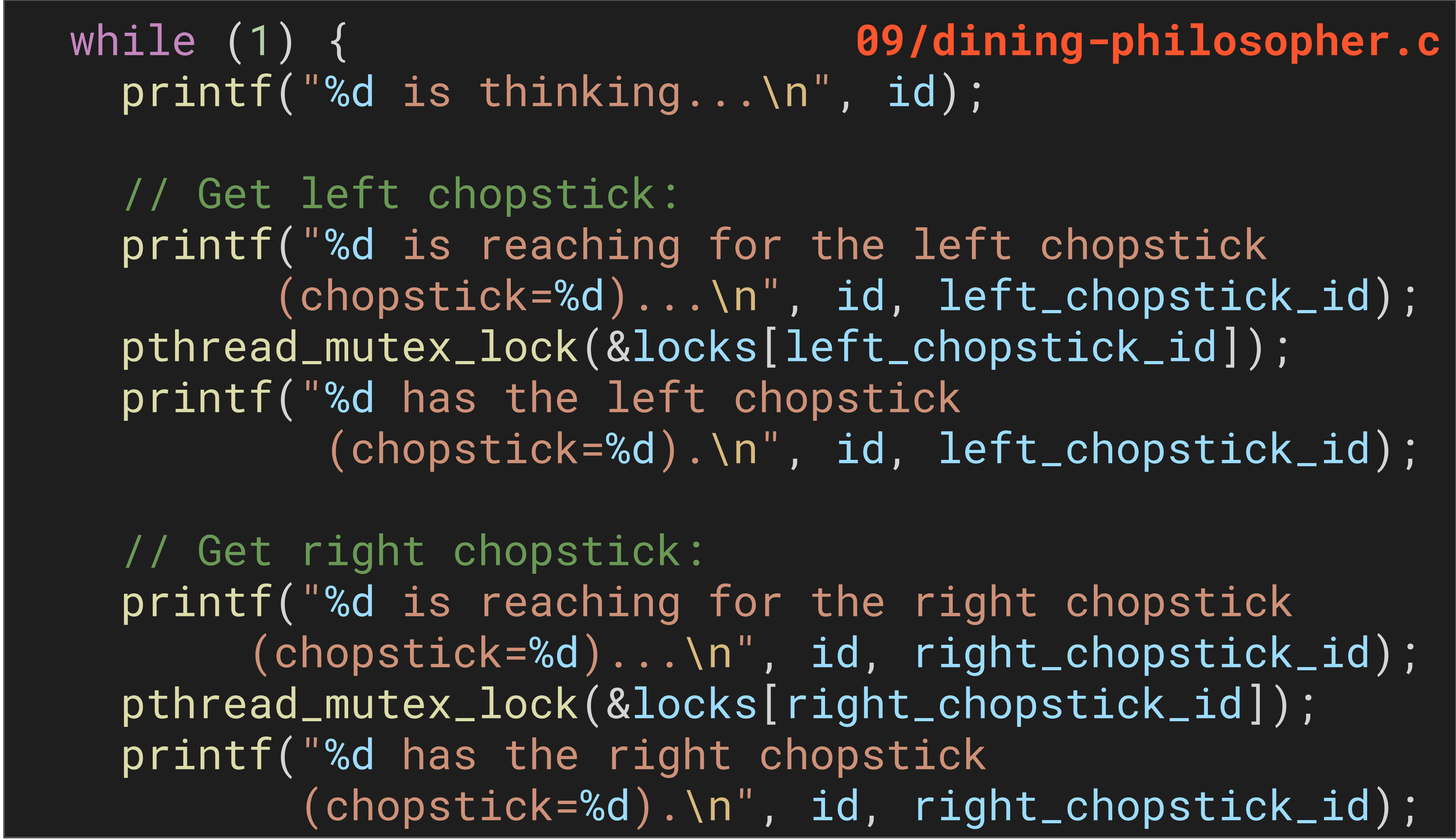
Critical Section

Dining Philosophers



Dining Philosophers





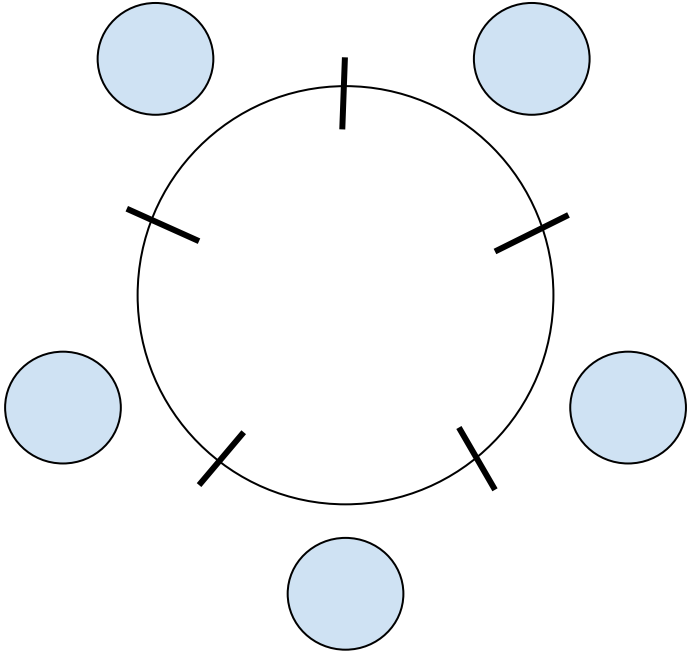
09/dining-philosopher.c

```c
  while (1) {
    printf("%d is thinking...\n", id);

    // Get left chopstick:
    printf("%d is reaching for the left chopstick
          (chopstick=%d)...\n", id, left_chopstick_id);
    pthread_mutex_lock(&locks[left_chopstick_id]);
    printf("%d has the left chopstick
            (chopstick=%d).\n", id, left_chopstick_id);

    // Get right chopstick:
    printf("%d is reaching for the right chopstick
         (chopstick=%d)...\n", id, right_chopstick_id);
    pthread_mutex_lock(&locks[right_chopstick_id]);
    printf("%d has the right chopstick
           (chopstick=%d).\n", id, right_chopstick_id);
```

09/dining-philosopher.c

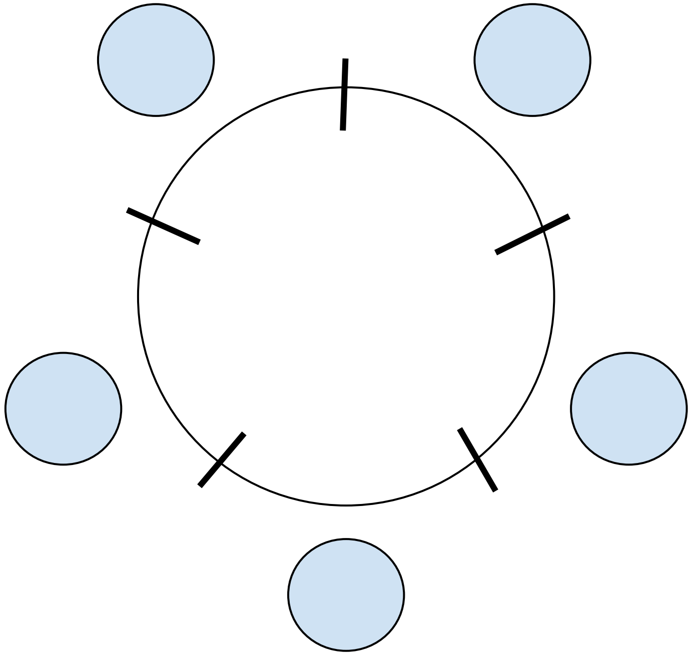
```c
    // Eat:
    printf("%d is eating... 🍱🥢\n", id);

    // Release chopsticks:
    printf("%d is returning their chopsticks
                (chopsticks: %d, %d)...\n", id,
                left_chopstick_id, right_chopstick_id);
    pthread_mutex_unlock(&locks[right_chopstick_id]);
    pthread_mutex_unlock(&locks[left_chopstick_id]);
  }
```

```
while (1) {
  think();
  lock_left(&mutex);
  lock_right(&mutex);
  eat();
  release_right(&mutex);
  release_left(&mutex);
}
```

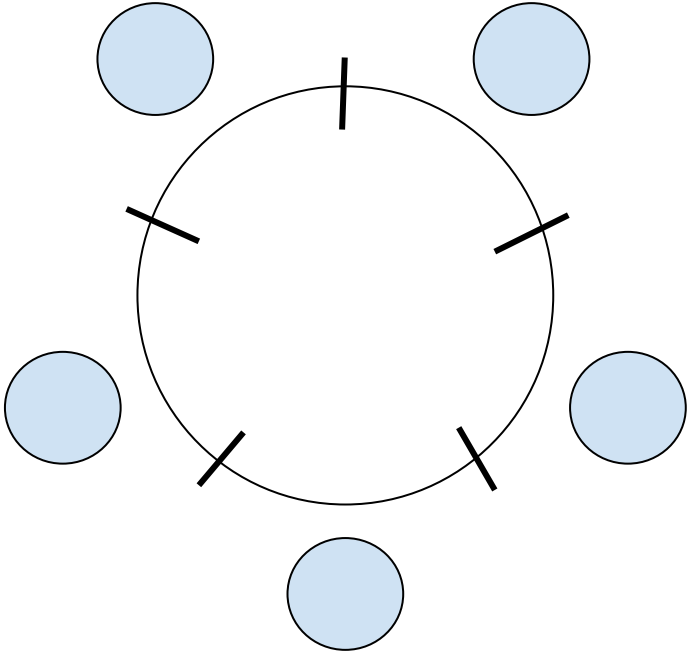


Four Necessary Conditions of Deadlock (“Hoffman Conditions”):

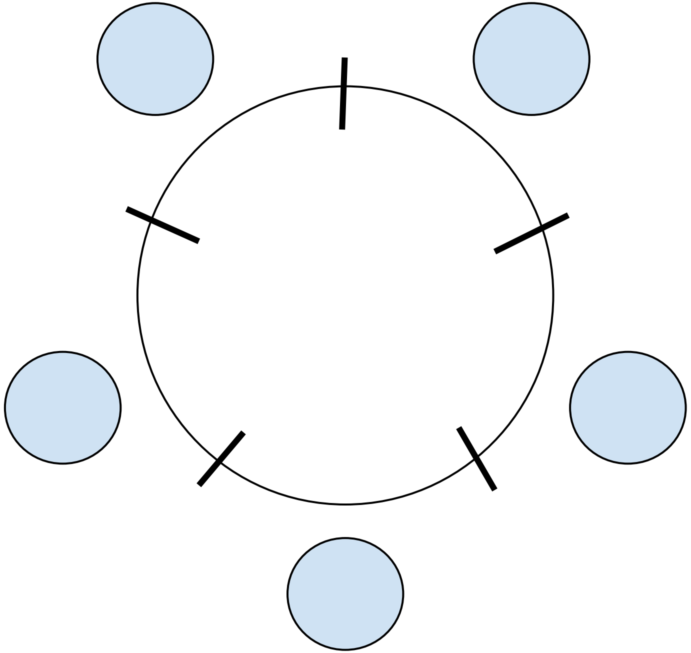
Mutual Exclusion



Circular Wait



Hold and Wait



No Preemption