

**Synchronization: Three Techniques**

For C-level synchronization, there are three constructs that we have available to help us synchronize access to **critical sections**:

**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**

```

5 pthread_mutex_t lock;
6 int ct = 0;
7
8 void *thread_start(void *ptr) {
9     int countTo = *((int *)ptr);
10
11     int i;
12     for (i = 0; i < countTo; i++) {
13         pthread_mutex_lock(&lock);
14         ct = ct + 1;
15         pthread_mutex_unlock(&lock);
16     }
17
18     return NULL;
19 }
```

**Q:** What happens when we run this code now?

...and the performance?

**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/producer-consumer.c**

```

11 int things[THINGS_MAX];
12 int things_ct = 0;
13
14 void *producer(void *vptr) {
15     while (1) {
16         pthread_mutex_lock(&lock);
17
18         // Cannot produce until there's space:
19         while (things_ct >= THINGS_MAX) {
20             pthread_cond_wait(&cond, &lock);
21         }
22
23         // Produce a thing:
24         things[things_ct] = rand();
25         printf("Produced [%d]: %d\n", things_ct, things[things_ct]);
26         things_ct++;
27
28         // Signal any waiting consumers:
29         pthread_cond_broadcast(&cond);
30
31         pthread_mutex_unlock(&lock);
32     }
33 }
```

### 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.

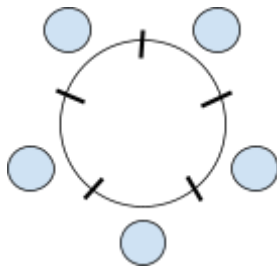
### Critical Sections

We know that critical sections require exclusive access to a resource. We also know locking a resource is computationally expensive. However, are there other concerns?

### The Dining Philosophers

Imagine five philosophers and five chopsticks at a circular table. Each philosopher has two states: **eating** and **thinking**:

- When a philosopher is thinking, she holds no chopsticks.
- When a philosopher starts the process of eating, she must take the chopstick to her left, then her right, and then begin eating.



**Q:** Using the strategy described above (take left, take right, then eat), what happens over a long period of time?

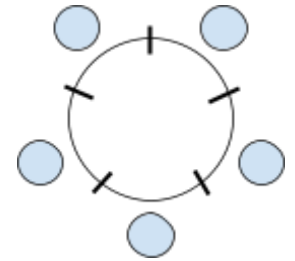
See Lecture Code: 09/dinning-philosophers.c

### Deadlock:

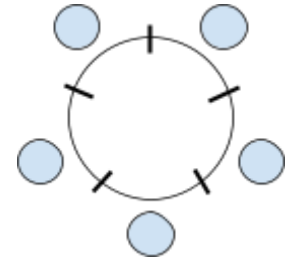
- Definition:

- Four **necessary** conditions of deadlock:

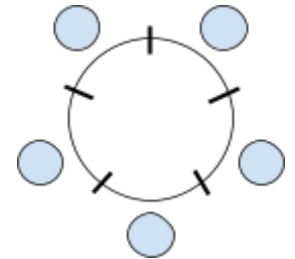
1)



2)



3)



4)

