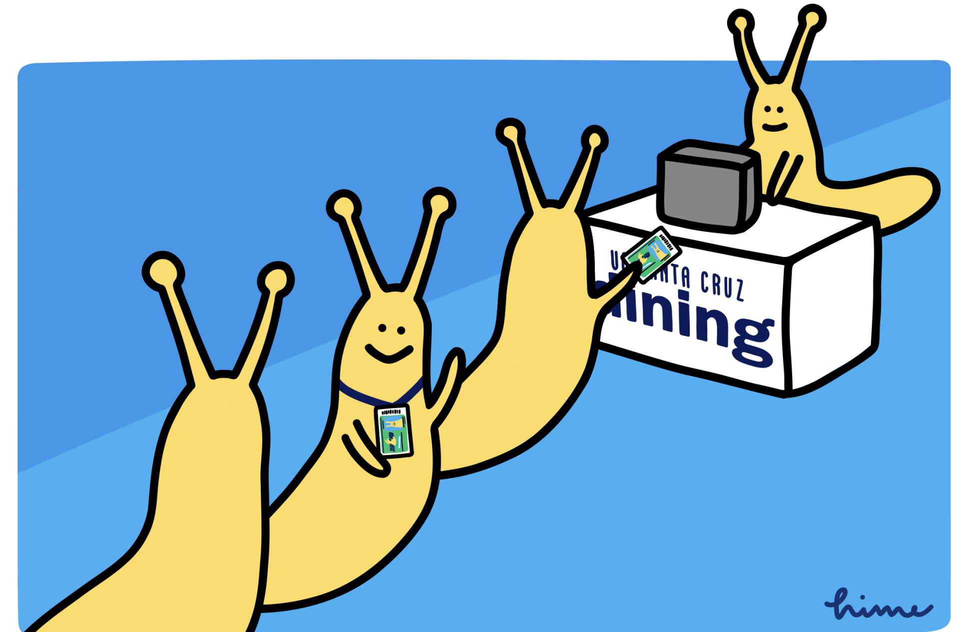
Project 3: Slug Dining

CSE 130 Principles of Computer Systems Design

Spring 2023



Project 3 is out!

- Use synchronization primitives to implement complex specification
- Due 5/14
- GitHub Classroom to create a repo, submit on Gradescope

Slug Dining

- You are hired as a system manager at the UCSC dining hall.
  - Write `dining.c` that works as a dining hall reception.
- Each dining has a `capacity`: the maximum number of students that can be inside the dining hall at a time.
- The cleaning provider often comes in to clean the dining hall
  - They use chemicals so they can only clean when no students are present

API

- `dining_t`: `struct` that holds necessary variables.
- `dining_t* dining_init(int capacity)`: constructor for `dining_t`.
- `void dining_student_enter(dining_t* dining)`: Called when a student tries to enter.
    - Blocks if the dining hall is full or cleaning is taking place.
- `void dining_student_leave(dining_t* dining)`: Called when a student leaves.
- `void dining_cleaning_enter(dining_t* dining)`: Called when the cleaning providers comes in.
    - Blocks if there is a student or cleaning is already taking place.
- `void dining_cleaning_leave(dining_t* dining)`: Called when cleaning is complete
- `void dining_destroy(dining** ptr)`

Example 1

```
dining_t* d = dining_init(3);

dining_student_enter(d); // student 1
dining_student_enter(d); // student 2
dining_student_enter(d); // student 3

// cannot enter so this blocks
dining_student_enter(d); // student 4

// on a different thread
dining_student_leave(d); // student 1 leaves, allowing student 4 to enter
dining_student_leave(d); // student 2
dining_student_leave(d); // student 3
dining_student_leave(d); // student 4

dining_destroy(&d);
```

Example 2

```c
dining_t* d = dining_init(3);

dining_student_enter(d); // student 1

// this blocks
dining_cleaning_enter(d);

// on a different thread
dining_student_leave(d); // student 1 leaves. cleaning starts.

// cleaning in progress; cannot enter
dining_student_enter(d); // student 2

// on a different thread
dining_cleaning_leave(d); // cleaning is done. student 2 can enter.

dining_student_leave(d);

dining_destroy(d);
```

Extra credit (20 points)

- A naive implementation allows students to enter even if the cleaning provider is waiting.
  - If new students constantly enter the dining hall, the cleaning provider will have to wait indefinitely.
- Change your code so that cleaning provider does not have to wait indefinitely. Assume that students leave after a reasonable amount of time.
- (Work on this if you're absolutely sure about the required part)

Lock (Mutex)

- Provides mutual exclusion between threads
  - If one thread is in the critical section, it excludes the others from entering until it has completed the section
- Either locked or unlocked
- Allows only one thread to acquire a lock

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&lock);
balance = balance + 1;
pthread_mutex_unlock(&lock);
```

Condition Variable

- Mutex isn't powerful enough in some situations

```
int balance = 0;

void renter() {
  int salary = work_hard();
  balance += salary;
}

void landlord() {
  while(!(balance >= 1200)) {
    // waiting...
  }
  balance -= 1200;
}
```

Condition Variable

Mutex?

```
int balance = 0;
pthread_mutex_t m;

void renter() {
  int salary = work_hard();
  pthread_mutex_lock(&m);
  balance += salary;
  pthread_mutex_unlock(&m);
}

void landlord() {
  pthread_mutex_lock(&m);
  while(!(balance >= 1200)) {
    // ?
  }
  balance -= 1200;
  pthread_mutex_unlock(&m);
}
```

Condition Variable

```
int balance = 0;
pthread_mutex_t m;

void renter() {
  int salary = work_hard();
  pthread_mutex_lock(&m);
  balance += salary;
  pthread_mutex_unlock(&m);
}

void landlord() {
  pthread_mutex_lock(&m);
  while(!(balance >= 1200)) {
    pthread_mutex_unlock(&m);
    // wait for renter to deposit
    pthread_mutex_lock(&m);
  }
  balance -= 1200;
  pthread_mutex_unlock(&m);
}
```

Condition Variable

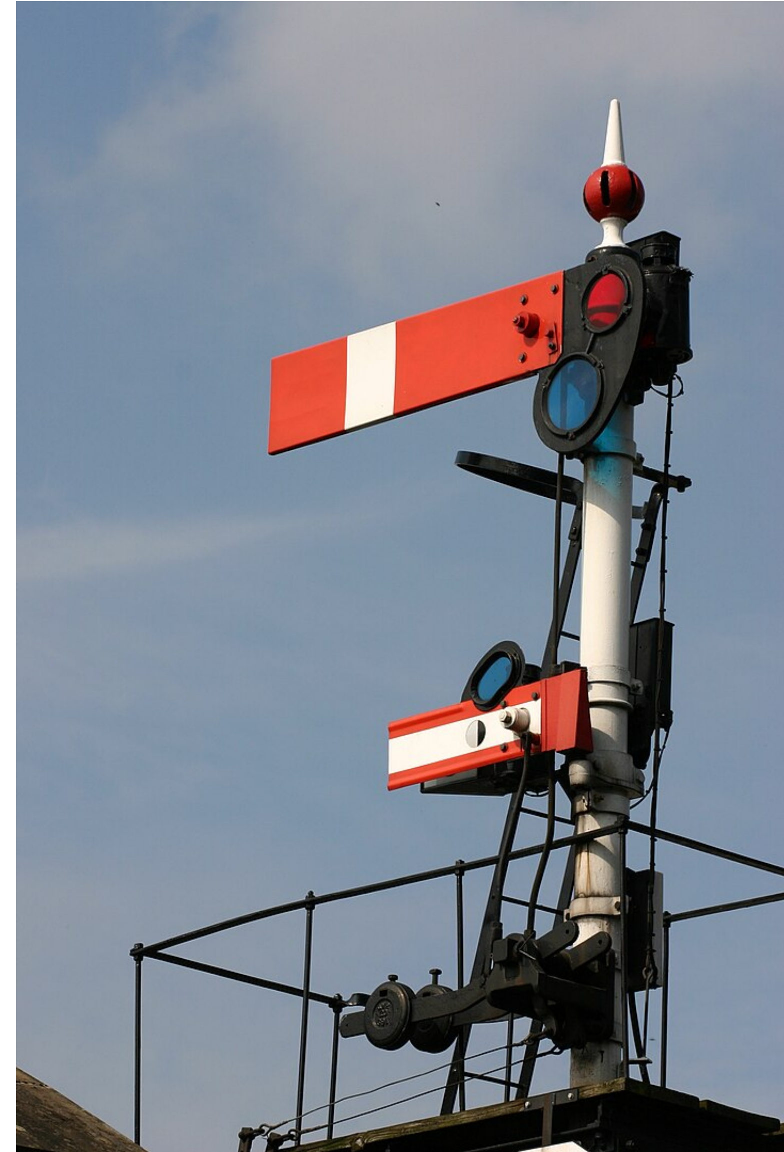
```c
int balance = 0;
mutex_t m;
pthread_cond_t c;

void renter() {
  int salary = work_hard();
  pthread_mutex_lock(&m);
  balance += salary;
  pthread_mutex_unlock(&m);
  pthread_cond_signal(&c);
}

void landlord() {
  pthread_mutex_lock(&m);
  while(!(balance >= 1200)) {
    pthread_cond_wait(&c, &m); // unlock -> wait for signal -> lock
  }
  balance -= 1200;
  pthread_mutex_unlock(&m);
}
```

`pthread_cond_signal` sends a signal to one thread. But it is possible that, because of the change the thread made, more than one thread can unblock. Use `pthread_cond_broadcast` to send signals to all the threads waiting on the condition variable.

```
int balance = 0;
mutex_t m;
pthread_cond_t c;

void renter() {
  int salary = work_hard();
  pthread_mutex_lock(&m);
  balance += salary;
  pthread_mutex_unlock(&m);
  pthread_cond_broadcast(&c); // notify both landlord and IRS
}

void landlord() {
  pthread_mutex_lock(&m);
  while(!(balance >= 1200)) {
    pthread_cond_wait(&c, &m);
  }
  balance -= 1200;
  pthread_mutex_unlock(&m);
}

void irs() {
  pthread_mutex_lock(&m);
  while(!(balance >= 500)) {
    pthread_cond_wait(&c, &m);
  }
  balance -= 500;
```

Semaphore

- An object with an integer value
- Two operations:
  - Down (Wait, P): Decrement the value, block if 0
  - Up (Post, V): Increment
- Users specify the initial value
  - If initialized to two, works as a lock



Semaphore

```
sem_t sem;
int ret;
int count = 2;

sem_init(&sem, 0, count);

sem_wait(&sem); // -> 1
sem_wait(&sem); // -> 0
sem_wait(&sem); // -> -1? block

// on a different thread
sem_post(&sem); // unblock ↑
```