Name: _____ CruzID: _____

CSE 130 Midterm 2

Spring 2022

General Instructions: Welcome to the second midterm of CSE 130! You have 65 minutes to complete the exam. There are 15 multiple choice questions and 2 free response questions. Read the questions carefully and budget your time accordingly.

- **You must sign the academic integrity pledge on this page for us to grade your exam**. Failure to do so will result in a score of 0.
- **Place your name and CruzID at the top of each page in the blank provided**. We cannot promise that we will be able to grade the questions on any pages that are missing a name/CruzID.
- **All of your work and your answers must be submitted on these pages.** For each multiple choice question, fill in the bubble of the answer that you think is correct. For the free response questions, write your answers in the space provided.
- **Each multiple choice question has a single answer.**
- **This exam is closed-book, closed note.** You may not use any electronic devices. Your proctor will assume that you are violating the academic integrity pledge if you are using an electronic device.

Good Luck! You can do this :)

Academic Integrity: As a member of the UCSC community, I have an explicit responsibility to foster an environment of trust, honesty, fairness, respect, and responsibility. I understand that this exam is to be completed entirely on my own. I have neither given nor received unauthorized aid on this exam, nor have I concealed any violations of Academic Integrity. I understand that violation of this pledge will result in a 0 on the exam, removal from this course, and/or a disciplinary action at the university level.

Name: _____

CruzID: _____

Signature: _____

Date: _____

Part II. Free Response Questions

Eugebe Food Services. Following the success of Eugebe Storage Systems, popularized by SLUG's surprisingly buggy `slug_write_async` interface, Eugene has decided to pursue his true passion: food services. Eugebe Food Services is breaking into the industry by operating a food truck that will serve Swedish-style meatballs on the UCSC campus. You have been hired as Eugene's Chief Operating Officer—your task will be to determine the procedures that govern how food will be served to customers.

1. (25 points) Eugebe Food Services begins with a fleet of kitchen-less food trucks that are designed for two servers. Each truck has two checkout windows, two vats of sauce, and a single tray for meatballs[1]. Each server should repeatedly dish four meatballs onto a plate, add sauce, and serve the plate to the next customer. However, the servers will spill if they both try to dish meatballs at the same time. We describe the API for each server with additional details below. Your task is to prevent all spillage throughout the three parts of this question: 1(a)–1(c).

Function	Description
`Meatball* dish_meatball()`	Get a meatball, or NULL if they are all gone.
`void add_sauce(Meatball *[])`	Add sauce to meatballs.
`void serve(Meatball *[])`	Give meatballs to customer.

[1] Eugene cooks the meatballs at a central location and places them into the meatball tray before each day.

(a) (10 points) The original procedure for the two workers is shown below. Each server calls `server` with their id, either 0 or 1, as an argument. The truck should close after it runs out of meatballs; it will close after both servers return from `server`. What problem(s), besides busy-waiting, does it possess? Provide a concrete example of each issue.

```
1 // Shared State
2 int turn = 0;
3
4 void server(int i) {
5   while (1) {
6     Meatball *plate[4];
7     while (turn != i) {}
8
9     for (int i = 0; i < 4; ++i) {
10        plate[i] = dish_meatball();
11        if (plate[i] == NULL)
12            return;
13    }
14
15    turn = 1 - turn;
16    add_sauce(plate);
17    serve(plate);
18  }
19 }
```

[Extra Page for (a)]

(b) (10 points) To resolve some of the issues with the prior algorithm, your team proposes the following suggestion. Like the past algorithm, each server calls `server` with their id, either 0 or 1. What problem(s), besides busy-waiting, does this version possess? Provide a concrete example of each issue.

```
1 // Shared State
2 int turn;
3 int interest[2];
4
5 void server(int i) {
6     while (1) {
7         Meatball *plate[4];
8         int other = 1 - i;
9
10        interest[i] = 1;
11        turn = other;
12
13        while (turn == other && interest[other]) {}
14
15        for (int i = 0; i < 4; ++i)
16            plate[i] = dish_meatball();
17
18        interest[i] = 0;
19
20        for (int i = 0; i < 4; ++i)
21            if (plate[i] == NULL)
22                return;
23
24        add_sauce(plate);
25        serve(plate);
26    }
27 }
```

[Extra Page for (b)]

(c) (5 points) The team implements fixes for the problem(s) you found in part b. Eugebe Food Services is interested in moving to food trucks with even more servers. Describe the design of a solution that would work for an arbitrary number of threads in a few sentences. What challenges would it need to overcome?

2. (30 points) Eugebe Food Services has decided to expand to significantly larger food trucks. The new trucks have kitchens to produce meatballs when the trucks are deployed (assume an infinite supply of meatballs to cook). Each truck will have c cooks, each of whom should repeatedly cook a meatball and place it into the truck's single tray. Additionally, each truck will have s servers, each of whom should repeatedly serve customers by dishing a plate of meatballs and adding sauce (assume an infinite queue of customers). If a server is dishing a meatball when a cook adds one to the tray, then the server will be burnt. Additionally, a server can be burnt if they reach for meatball in the tray but there are none there. Finally, if two cooks try to place meatballs into the tray at the same time, then they can be burnt. We outline the cook and server interface below.

Caller	Function	Description
server	`Meatball* dish_meatball()`	Dish a single meatball.
server	`void add_sauce(Meatball* [])`	Add sauce to meatballs.
cook	`void add_to_tray(Meatball*)`	Add meatball to tray.

Additionally, cooks and servers can synchronize using locks, condition variables, and semaphores. One notable difference from what we've discussed in class is a `broadcast` function that signals all of the servers/cooks waiting on a condition variable. Specify `Sem s(n)` to initialize a semaphore, `s`, to the value `n`; `Lock l` to initialize a lock, `l`; and `CV c` to initialize a CV, `c` .

Function	Description
`void acquire(Lock l)`	Acquire lock `l`.
`void release(Lock l)`	Release lock `l`.
`void signal(CV c)`	Signal on condition variable `c`.
`void broadcast(CV c)`	Broadcast on condition variable `c`.
`void wait(CV c, Lock l)`	Wait on condition variable `c` using lock `l`.
`void down(Sem s)`	Down on semaphore `s`.
`void up(Sem s)`	Up on semaphore `s`.

(a) (15 points) Implement a solution for dishing meatballs that supports c cooks, s servers, and a tray with maximum size t. Your solution should prevent spillage (i.e., no two servers should be using the tray at once) and burns (i.e., no cook should place a meatball into the tray if a server is dishing or another cook is placing a meatball into the tray, and a server should only be dishing if there is a meatball to dish). Each time a server needs to prepare a plate of meatballs, they will call your `server` function. The server function should place exactly 4 meatballs in the `plate` argument, which should not "get cold" (i.e., a server should ensure that they can dish 4 meatballs before dishing any meatballs), and then add sauce to them. Each time a cook is ready to place a meatball into the tray, they will call your `cook` function, which should safely place the `mb` argument into the tray. Place any shared state in the "shared state" region below and provide implementations for you `server` and `cook` functions on the following page.

Solution:

```
1 // shared state
2 int num_meatballs = 0;
3 Lock l;
4 CV empty, full;
5
6 void server(Meatball *plate[4]) {
7   acquire(l);
8   while (num_meatballs < 4)
9     empty.wait();
10
11  for (int i = 0; i < 4; ++i)
12    plate[i] = dish_meatball();
13  num_meatballs -= 4;
14  full.signal();
15  release(l);
16
17  add_sauce(plate);
18 }
19
20 void cook(Meatball *mb) {
21  acquire(l);
22  while (num_meatballs == v) {
23    full.wait();
24  }
25
26  add_to_tray(mb);
27  if (num_meatballs == 4) {
28    empty.signal();
29  }
30  release(l);
31 }
```

(b) (15 points) Eugebe Food Services developed a new form of ladle for dishing meatballs that prevent servers from spilling if they both reach into the tray at the same time. Implement a solution for dishing meatballs that supports c cooks, s servers, and a tray with maximum size t. Your solution should prevent burns (i.e., no cook should place a meatball into the tray if a server is dishing or if a different cook is placing meatballs into the tray, and a server should only be dishing if there is a meatball to dish). Each time the server prepares a plate of meatballs, they will call your `server` function. The server function should place exactly 4 meatballs in the `plate` argument, which should not "get cold" (i.e., a server should ensure that they can dish 4 meatballs before dishing any meatballs), and then add sauce to them. Each time a cook is ready to place a meatball into the tray, they will call your `cook` function, which should safely place the `mb` argument into the tray. One additional difference between this part and the previous one—each cook should have priority when placing meatballs into the tray so that they do not become too cold. Place any shared state in the "shared state" region below and provide implementations for you `server` and `cook` functions on the following page.

Solution:

```
1 // shared state
2 int num_meatballs = 0, num_servers = 0, num_cooks = 0,
3 Lock l;
4 CV server_cv, cook_cv;
5
6 void server(Meatball *plate[4]) {
7   acquire(l);
8
9   while ((num_cooks > 0 && num_meatballs < t ) ||
10           num_meatballs < 4 )
11     wait(server_cv, l);
12
13  num_meatballs -= 4;
14  num_servers += 1;
15  release(l);
16
17  for (int i = 0; i < 4; ++i)
18    plate[i] = dish_meatball();
19
20  acquire(l)
21  num_servers -= 1;
22  signal(cook_cv);
23  release(l)
24
25  add_sauce(plate);
26 }
27
28 void cook(Meatball *mb) {
29  acquire(l);
30  num_cooks += 1;
31  while (num_servers > 0 || num_meatballs == t) {
32    signal(server_cv);
33    wait(cook_cv, l);
34  }
35
36  add_to_tray(mb);
37  signal(cook_cv);
38  signal(server_cv);
39  num_cooks -= 1;
40  release(l);
41 }
```

(c) (10 points) **Extra Credit**: Suppose that there is only a single cook (i.e., $c = 1$). Implement the same problem as the previous part, but ensure that neither the cook nor any servers experience starvation.

Name: _____ CruzID: _____

Solution:

```
1 // shared state
2 int num_servers = 0, num_produced = 0, num_consumed = 0,
      ticket = 0;
3 Lock l;
4 CV server_cv, cook_cv;
5
6 void server(Meatball *plate[4]) {
7   acquire(l);
8   int my_ticket = ticket++;
9
10  while (num_produced < my_ticket * 4)
11    wait(server_cv, l);
12
13  num_consumed += 4;
14  num_servers += 1;
15  release(l);
16
17  for (int i = 0; i < 4; ++i)
18    plate[i] = dish_meatball();
19
20  acquire(l)
21  num_servers -= 1;
22  signal(cook_cv);
23  release(l)
24
25  add_sauce(plate);
26 }
27
28 void cook(Meatball *mb) {
29  acquire(l);
30  while (num_servers > 0 ||
31         num_produced - num_consumed == t)
32    wait(cook_cv, l);
33
34  add_to_tray(mb);
35  num_produced += 1;
36  broadcast(server_cv);
37
38  release(l);
39 }
```