CSE344 – System Programming - Signal Handling, Midterm Report

Ahmed Semih Özmekik, 171044039
May 10, 2020

Abstract

Detailed explanation of solutions for problems encountered in homework and graphs.

1 Problem

First of all, let's try to get to know the elements of our problem before we dive into it. At first glance, we see that there are three different actors. Our goal is simply to make the interactions of these there actors healthy with each other. To do that, we need to consider the rooms and objects that they all use in common. After creating the idea of allocating the resources and contexts, common areas of these three actors and determining the boundaries in theory, we will write the code. Therefore, we realize that it will not matter how many of an actor is in our scenario, and we will develop our idea that each process will work properly with both a sample of its own kind and the processes of other actor kinds.

Let's start with the supplier actor first. The scenario of the supplier actor supplying supply to cook is actually simulated only with the role of bringing plates. But one important thing is that the supplier and student roles do not interact directly with each other. Although student and supplier have to communicate with cook at a common point, they are unaware of each other. In this case, the actor who actually has to communicate with the two actors is just the cook actor. Perhaps this makes our job a little easier.

If we construct the two synchronization situations independently and make both successful, our system will work properly if we combine these two scenarios. So first of all I consider it useful to take a situation and take care of it first. Perhaps, in this way, we may encounter some problems that are likely to happen for the other situation at the stage of realization of the system and prepare for the other situation. So for now, let's just consider the supplier and cook actors and only and only their interactions.

There are K places in the kitchen for the supplier to put. So if there are as many plates of K in this place, the supplier will wait. On the other hand, cook is trying to get these plates. It is immediately noticeable that this is just a simple producer-consumer problem, as in the other case, which we consider independent of the current situation. But the only problem is that there are more than one producer and consumer process in the student-cook situation. If we are to adjust this, since the producer will be cooks, all cooks have to work both among themselves and with all other students. But now there is only one producer and more than one consumer (cook). So the supplier's workflow will be just like this: While he (supplier) is filling the kitchen with plates, he will wait for the place to be emptied..

2 Solution

Firstly we will have 2 shared memory samples. This represents 2 different rooms. After all, two different processes from any actor should not enter these rooms at the same time. So we will use a semaphore for each shared memory instance to set the synchronization of the entrance.

Of course, we will not deal primarily with the other situation mentioned for now. If we only deal with the first situation, we have to create a kitchen room. After creating this, we will define one semaphore to control the entrance and exit of the kitchen room. Let's not forget that the suppliers and all the cooks will have room information and this control semaphore. Then we will use one more semaphore to regulate our logic in the program flow.

2.1 Design Solution

First of all, while considering the design options on the early stages, i.e memory sharing stage. I encountered two design options. The first is to create a different **struct** (i.e. shared memory structure) for each context and transfer it between certain processes. The second is to create a generic structured shared memory structure and ensure that all actors and all processes use this generic **struct** structure. There's a trade-off between these two, of course.

Since I don't follow generic structure when I use the first one, I can be as specific as I want and I can just create that variable exactly what I need. But the disadvantage of this situation is that I will have to design a set of non-generic and different functional groups (for construction and destruction) for each structure. In the second case, I will decide on a certain number of semaphores and variable numbers. And sticking to this, I will only have a generic set of functions for this structure. And I will stick to this structure in all contexts and actors. But the disadvantage of this situation is that in some contexts, if I do not need it, it will be a semaphore that I will not use it and takes up space in memory.

But when I started writing, I noticed that, in a context, at most 2 or 3 semaphores not used. In this case, I decided to sacrifice 2 or 3 semaphore areas for a nice generic and orderly structure, and a better code reusability.

Since the sizes of the rooms are not certain from the beginning, they can change according to the parameters, it is considered as a separate shared-memory. In this way, I created and shared exactly how much space we need.

After all this, I had a structure like this.

```c

struct ipc_item
{
    int *room;
    int *peek;
    sem_t *room_mutex;
    sem_t *peek_mutex;
    sem_t *empty; // empty slot in room.
    sem_t *full;  // full slot in room.
};

struct ipc_shared_mem
{ // FOOD_KIND == 3.
    int peek;
    sem_t room_mutex;
    sem_t peek_mutex;
    sem_t empty[FOOD_KIND];
    sem_t full[FOOD_KIND];
};
```

Then, of course, when we ask for a reference for each field from the actor we want, we will do this with the necessary key. The shared memory keys and an example usage are as follows.

```c
 // shared memory names.
const struct ipc_key KITCHEN = {"k-room", "k-semaphore"};
const struct ipc_key COUNTER = {"c-room", "c-semaphore"};
const struct ipc_key TABLE = {"t-room", "t-semaphore"};
const struct ipc_key SUPERVISOR = {"s-room", "s-semaphore"};

void cook(int nth)
{
    // get resources.
    // shared resources between children cooks and other actors.
    struct ipc_item k = ipc_open(KITCHEN); // kitchen communication.
    struct ipc_item c = ipc_open(COUNTER); // counter communication.
    ...
    ...
    ...
}
```

Some of the things I shared focused on design problems and solutions. Now we can move on to the main problems of the assignment, that is, the solutions that contain solutions for synchronization.

2.2 Solution for Kitchen

For the kitchen context, there are two actors. These actors are, cooks and supplier. It is important to remember that we have contexts with different actors such as kitchen, cook and tablet. The actor processes will have memory and semaphores that we will share. The real synchronization problem between the kitchen and the cook is actually an almost simple producer-consumer problem. But the only difference is this. Although the producer, the suppler, supplies it randomly, the cooks have to get it according to a certain order (according to the need in the counter room). The supplier will produce it randomly, and there is a **2LM + 1** place in the kitchen. In this case, we see that for the kitchen, when the kitchen room is full, there is at least one of each supply type inside. If we make every cook to enter only when the kitchen room is fully occupied, we face a problem with the last **2LM** supplies in room, that is, with the last supplies, because when the cook is about to be finished, the cook cannot transmit the last supplies to the students since it's forbidden. But of course, we can use one more extra semaphore and give a green light to cooks, indicating that supplier is on delivery of last supplies. but in any case of this solution, there is an inefficiency. Because the supplier only allows cooks when it is full. But perhaps, even in many cases, it is very likely that the food which a cook need is in the kitchen.

Here I chose a solution as follows. As can be seen from the structure above, there is a semaphore for each food. In this case, for example, a cook who wants to bring desert will go to kitchen and get a desert if there is a desert, otherwise he will block himself. So there is a full semaphore for each food. And when explaining cook, it will be seen in more detail that the food selection order there is there. It will be seen that the cooks decide in advance what food they will take from the kitchen. They then go to the kitchen to get it. In this case, if the food they want to take (only and only that food) is in the kitchen, they take it and put it on the counter. If it's not on the kitchen, they will block execution. Here is a sneak peak to the solution taken from code.

```
supplier():
{
    ...
    put_item(k.room, slot, food);
    printf("\nThe supplier delivered %s - after delivery: kitchen items %s", FOODS[food],
    total_str(k.room, args.K, str));
    spost(k.room_mutex);
    spost(&k.full[food]);
    ...
}

cook()
{
    ...
    printf("\nCook %d is going to the kitchen to wait for/get a plate - kitchen items: %s",nth,
    total_str(k.room, args.K, str));
    // kitchen room entrance
    swait(&k.full[food]);
    ...
}
```

In this way, no cook waits while he has a food kitchen he wants. And it is likely that this problem cannot be approached with a more efficient solution than this. Because there is something certain in the assignment that the supplier will supply it randomly. The worst case in this theory is that all of the **2LM** supply produced by the supplier contains only two kinds of food. And let's say there is no desert in the kitchen at any time. In our solution, when the desert is produced, the supplier posts the semaphore (as soon as it is produced), waking a desert waiting cook. So everything happens without losing time. This was the solution we brought to the synchronization in the kitchen context.

2.3 Solution for Counter

Since the bonus part of the assignment is also done, the actors in this section are as follows for me: Cooks, Students, Supervisor. Considering the bonus party, there are 2 main problems in this section. But first I will consider our first major problem, ignoring the bonus part. The problem is, as soon as a student enters the counter, there must be 3 kinds of food there. And the student will exit, only after taking from each kind of food.

If there was only one cook working in the system, we would say that, starting from a determined food, take a different food in the kitchen and put it on the counter in each iteration. So we would make sure that we always put a different food in order. But we know that there are more than one cook. In this case, even if we start different kinds of food in order to have different foods on the counter, the order we set up from the beginning will be broken because we cannot decide which cook will work in which order, and the counter is unnecessarily filled with the same food type, therefore we enter a deadlock. So there is something definitive. The cooks have to work in coordination with each other on which food they take to the counter.

I came up with a solution for this coordination. As I just said, if there was only one cook, we would have solved the work by iterating in sequence and filling the counter. If we adapt the solution here to more than one cook, we would have made a common iteration for all cook. Namely, we will specify the food that should be put on the counter in a shared variable. Every cook will look from here and find out what food to bring. And after learning, this will increase the food one so that the cook, who will read this variable next, will bring it. In other words, an idle cook will learn what to bring here, and it will increase this variable so that the next cook will find out what it will bring. Below is a solution from a piece of code.

```


cook()
{
    ...
    // peek the needed for the student.
    swait(c.room_mutex);
    swait(c.peek_mutex);
    if (*c.peek >= FOOD_KIND)
        *c.peek = P;
    food = *c.peek;
    ++*c.peek;
    spost(c.peek_mutex);
    spost(c.room_mutex);
    ...
}
```

There is a small problem that comes to mind in this solution, which is an emerging problem on where the cooks will stop. We simply understand this according to whether or not the delivery of that food kind is made as required.

Now we are sure that every cook is going to get different foods. However, by determining this, we are not sure that there are different kinds of foods on the counter. Because again, we do not know which row the cook will work. For example, we have a 4-size counter: **[desert, maincourse, desert, 0]**. We now know that if there are 4 cooks in total, 3 of them bring something different, and one of them brings the same kind of food as one of these 3. Because then the c.peek variable will be toured, start from again. In this case, we need only soup for the counter. In order to fill the slot in the counter, we can only allow those who bring soup from 4 working cook. Others will cause deadlock. This was actually the main part of the problem. The solution I brought against this is just like the solution in the context of the kitchen.

The solution is as follows. Let's consider the difference of this problem from the previous problem. At a given time, a cook wants a certain food. Then he waits or continues with that food's semaphore. In this first problem, the cook, who is a consumer, becomes a producer, just like the supplier, in the second problem. But the only difference is that students need to dive in, not when any food is ready, but when all 3 are ready. Actually, this approach will not change for producer, cook, but students taking approach will change. But for cooks one very important thing is changing: They will not produce if there is a gap in the entire kitchen as in the supplier, but they will be able to put it only if there is free space for the food he is putting. Let's say the counter is like this, **[0, 0, 0, 0]**. So it's all empty. In this case, we only give one slot right **(size / FOODKIND)** for each food. If that semaphore is 0, the cook will block himself and it will not be able to put it there. If the number of slots is 3k + 1, the remainder from the food kinds (that is 3) is 1. In this case, we have the right to give one more slot to any food type. It does not matter to which food variety we will grant this right. After all, one of them can be put and full efficiency can be obtained from the counter in this way. If it is 3k + 2 the remainder will be 2 out of 3k rights. Then we can grant 2 more slots to one of the kinds, or we can grant two more of the slots to two of the kinds. Of course, it makes more sense to choose the second one because we want to increase the variety in the counter. If we choose the second one, we will get the highest efficiency from the counter by preventing a cook from waiting unnecessarily afterwards, by providing diversity already. When the counter size 3k, they will all have the right to put up to k.

As can be seen below, after cook taking a food from the kitchen, it proceeds as follows:" Do I have the right to put it on the counter from this type of food? If there is, I put it. Otherwise, I expect this right to be given again." And of course, the situation that brought this right back to him is that the student who entered the counter posted this semaphore. This means to tell cook that the student took that dish from the counter and that a new slot for that dish was given. Let's examine the cook side of this event shortly. Take a look at how the cook manages this process below.

```
cook()
{
    ...
    int food_delivered[FOOD_KIND] = {deliver, deliver, deliver};
    while(TRUE)
{     ...
    // peek the needed for the student.
    swait(c.room_mutex);
    swait(c.peek_mutex);
    if (*c.peek >= FOOD_KIND)
        *c.peek = P;
    food = *c.peek;
    ++*c.peek;
    spost(c.peek_mutex);
    spost(c.room_mutex);
    ....

    // kitchen room entrance. (can be seen above)
    ...
    ...

    // counter room entrance
    swait(&c.empty[food]);
    swait(c.room_mutex);
    slot = next(c.room, args.S, EMPTY);
    put_item(c.room, slot, food);
    --food_delivered[food];
    spost(c.room_mutex);
    spost(&c.full[food]);
 }
}
```

We are sure now that there will be at least one of every kind of food on the counter. Now someone has to take action according to this situation. If the bonus party had not been made, I would say a student process would take action directly by waiting for the cooks and then taking the foods etc. However, a supervisor is required, which determines which student will buy the food first, that gives the graduates this right first, and if they do not have a pending graduate, they give this right to the undergraduates.

The only difference between a graduate and an undergraduate student is the difference of who has priority, and all the actions they do are the same. Here we have a small inner context that includes supervisors and students, although not as great as the others.

2.3.1 Solution for priority of undergraduates and graduates

This requires a separate inter process communication review, since it is a separate context. In this case, I have defined a new actor. This actor, is a supervisor. The only thing this actor does is to understand that when the food is ready at the counter (which we discussed above) and signal a student to get in there and get a food. What are certain things? If a graduate student is waiting, he has the priority. Supervisor also needs to understand that there is a graduate among those waiting. But it doesn't have to be a graduate student at any time. If it does not exist, it should continue to serve the foods immediately whenever the counter the is ready. In this case, we cannot say to the supervisor, "first serve all graduates, then serve the undergraduates." We could say that, but in this case we would not use the counter efficiently. Because while all graduates are eating and not in line, undergraduates will have waited for them in vain.

My solution is as follows. First of all, a supervisor will wait for all foods to arrive on the counter within the logic we have explained above. Then, by checking the **peek** variable, he will know if there is a graduate in order. If this variable is zero, it means that no graduate is waiting in the line. Since this variable will accumulate, we can also understand if there are more than one graduate in the line. If there is a waiting graduate in line, a green light will be given to the semaphore allocated for them. If there is no waiting graduate in line, a green light will be given to a semaphore allocated only for underground deposits as well. I can show it as follows:

```c
...
for (int i = 0; i < deliver; ++i) // delivering 3 in a row.
    {
        printf("\nSupervising....");
        for (int i = 0; i < FOOD_KIND; ++i) // wait for all kinds of foods to be ready.
            swait(&c.full[i]);
        printf("\nGot foods!");
        swait(s.empty); // if someone, any student is waiting.
        swait(s.peek_mutex);
        int graduate = (*s.peek) ? TRUE : FALSE; // if any graduate is waiting on the line.
        if (graduate)
            --*s.peek;
        spost(s.peek_mutex);
        spost(&s.full[graduate]);
    }
    printf("\nSupervisor finished!");
    exit(EXIT_SUCCESS);
...
```

As you can see, it is waiting for the food to come first. Then it wakes up the necessary student. Finally, let's see what action the student took in this situation. A student states this immediately if he is graduated before blocking himself into a wait state. And then he goes on hold. If he is awakened, it means that the supervisor has given him a green light. He says that he gets one of the three kinds of food one by one and as soon as he gets the foods, he says to the cooks that the slot is become empty. And thus, we ensure the most efficient use of the counter. In other words, while a student is still getting his food, he informs the cook that he has taken his food and a certain slot has been emptied, and he allows another cook to put the next food in there while he is taking it. So cooks don't expect a student to take everything out first from the counter. Instead they keep producing. The only restriction here is that, only one student will be on the counter at any point. But this does not hold for the other cook side of the. Cooks keep producing the foods once a slot is available. Here is my solution in code wise:

```c
...
    printf("\nStudent %d is going to the counter (round %d) - # of students at counter: 1 and
    counter items %s", nth, i, total_str(c.room, args.S, str));
    swait(s.peek_mutex);
    if (graduate)
        ++*s.peek;
    spost(s.peek_mutex);
    spost(s.empty); // student on the line.
    swait(&s.full[graduate]);
    swait(c.room_mutex);
    for (enum plate p = P; p < FOOD_KIND; ++p)
    {
        int slot = next(c.room, args.S, p);
        get_item(c.room, slot);
        spost(&c.empty[p]);
        printf("\nStudent %d took one %s (round %d)", nth, FOODS[p], i);
    }
    spost(c.room_mutex);
    // counter room exit.
...
```

2.4 Solution for Table

This is the other context of the students. This is the other context of the students. Synchronization of this context is relatively easier than others. Simply put, if there is no idle table, he waits, if he gets to sit, it eats and gets up. Solution:

```
printf("\nStudent %d got food and is going to get a table (round %d) - # of empty tables :%d"
,nth, i, item_number(t.room, args.T, EMPTY));
// look for table.
swait(t.empty);
swait(t.room_mutex);
int slot = next(t.room, args.T, EMPTY); // next empty table.
put_item(t.room, slot, TRUE);           // sit down and eat.
printf("\nStudent %d sat at table %d to eat (round %d) - empty tables:%d",
 nth, slot, i, item_number(t.room, args.T, EMPTY));
put_item(t.room, slot, EMPTY); // stand up.
spost(t.room_mutex);
spost(t.empty); // leave the table.
printf("\nStudent %d left table %d to eat again (round %d) - empty tables:%d",
            nth, slot, i, item_number(t.room, args.T, EMPTY));
```

3 Plots

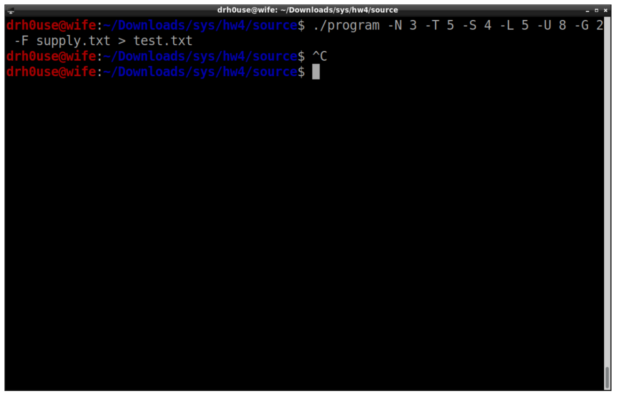
In this section, we will examine the change of dependent variables with graphs by changing some of our programs parameters.

A line in an output after running the program is considered a unit of seconds. I will change one parameter and examine the system while keeping the others constant. Then I will make an inference by examining the data on the results of this input. While collecting the data, I will consider the supply file we have as an independent variable and keep it constant.

Before I show the data and charts I have collected, I want to show my data collection strategy here. First of all, I direct the output of the program to a file like this.

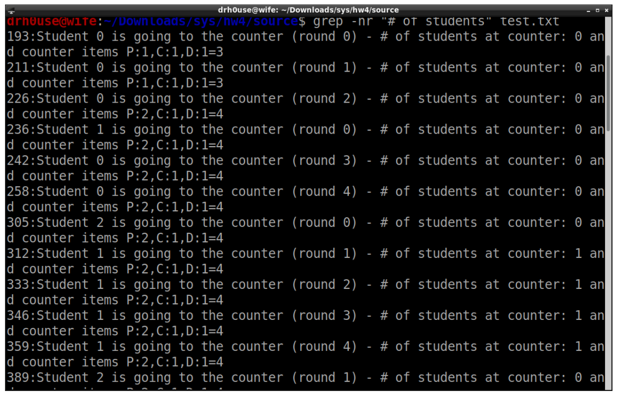
```
./program -N 3 -T 5 -S 4 -L 5 -U 8 -G 2 -F supply.txt > test.txt
```

Then I just take the part that interests me with the line numbers and write them on the chart. As I said, the number of lines is a unit of time for me.



```
drh0use@wife:~/Downloads/sys/hw4/source$ ./program -N 3 -T 5 -S 4 -L 5 -U 8 -G 2
 -F supply.txt > test.txt
drh0use@wife:~/Downloads/sys/hw4/source$ ^C
drh0use@wife:~/Downloads/sys/hw4/source$ 
```

Figure 1



```
drh0use@wife:~/Downloads/sys/hw4/source$ grep -nr "# of students" test.txt
193:Student 0 is going to the counter (round 0) - # of students at counter: 0 an
d counter items P:1,C:1,D:1=3
211:Student 0 is going to the counter (round 1) - # of students at counter: 0 an
d counter items P:1,C:1,D:1=3
226:Student 0 is going to the counter (round 2) - # of students at counter: 0 an
d counter items P:2,C:1,D:1=4
236:Student 1 is going to the counter (round 0) - # of students at counter: 0 an
d counter items P:2,C:1,D:1=4
242:Student 0 is going to the counter (round 3) - # of students at counter: 0 an
d counter items P:2,C:1,D:1=4
258:Student 0 is going to the counter (round 4) - # of students at counter: 0 an
d counter items P:2,C:1,D:1=4
305:Student 2 is going to the counter (round 0) - # of students at counter: 0 an
d counter items P:2,C:1,D:1=4
312:Student 1 is going to the counter (round 1) - # of students at counter: 1 an
d counter items P:2,C:1,D:1=4
333:Student 1 is going to the counter (round 2) - # of students at counter: 1 an
d counter items P:2,C:1,D:1=4
346:Student 1 is going to the counter (round 3) - # of students at counter: 1 an
d counter items P:2,C:1,D:1=4
359:Student 1 is going to the counter (round 4) - # of students at counter: 1 an
d counter items P:2,C:1,D:1=4
389:Student 2 is going to the counter (round 1) - # of students at counter: 0 an
```

Figure 2

```
$ grep -nr "# of students" test.txt

193:Student 0 is going to the counter (round 0) - # of students at counter: 0 and counter
211:Student 0 is going to the counter (round 1) - # of students at counter: 0 and counter
226:Student 0 is going to the counter (round 2) - # of students at counter: 0 and counter
236:Student 1 is going to the counter (round 0) - # of students at counter: 0 and counter
242:Student 0 is going to the counter (round 3) - # of students at counter: 0 and counter
258:Student 0 is going to the counter (round 4) - # of students at counter: 0 and counter
305:Student 2 is going to the counter (round 0) - # of students at counter: 0 and counter
312:Student 1 is going to the counter (round 1) - # of students at counter: 1 and counter
333:Student 1 is going to the counter (round 2) - # of students at counter: 1 and counter
346:Student 1 is going to the counter (round 3) - # of students at counter: 1 and counter
359:Student 1 is going to the counter (round 4) - # of students at counter: 1 and counter
389:Student 2 is going to the counter (round 1) - # of students at counter: 0 and counter
403:Student 2 is going to the counter (round 2) - # of students at counter: 0 and counter
412:Student 2 is going to the counter (round 3) - # of students at counter: 0 and counter
432:Student 2 is going to the counter (round 4) - # of students at counter: 0 and counter
442:Student 9 is going to the counter (round 0) - # of students at counter: 1 and counter
443:Student 5 is going to the counter (round 0) - # of students at counter: 2 and counter
444:Student 7 is going to the counter (round 0) - # of students at counter: 2 and counter
445:Student 4 is going to the counter (round 0) - # of students at counter: 3 and counter
446:Student 8 is going to the counter (round 0) - # of students at counter: 2 and counter
467:Student 9 is going to the counter (round 1) - # of students at counter: 4 and counter
469:Student 3 is going to the counter (round 0) - # of students at counter: 1 and counter
475:Student 3 is going to the counter (round 1) - # of students at counter: 5 and counter
477:Student 6 is going to the counter (round 0) - # of students at counter: 1 and counter
485:Student 9 is going to the counter (round 2) - # of students at counter: 6 and counter
494:Student 9 is going to the counter (round 3) - # of students at counter: 6 and counter
503:Student 9 is going to the counter (round 4) - # of students at counter: 6 and counter
525:Student 7 is going to the counter (round 1) - # of students at counter: 5 and counter
531:Student 7 is going to the counter (round 2) - # of students at counter: 5 and counter
543:Student 5 is going to the counter (round 1) - # of students at counter: 5 and counter
552:Student 3 is going to the counter (round 2) - # of students at counter: 5 and counter
566:Student 7 is going to the counter (round 3) - # of students at counter: 4 and counter
578:Student 7 is going to the counter (round 4) - # of students at counter: 4 and counter
583:Student 3 is going to the counter (round 3) - # of students at counter: 4 and counter
597:Student 5 is going to the counter (round 2) - # of students at counter: 4 and counter
604:Student 5 is going to the counter (round 3) - # of students at counter: 4 and counter
615:Student 3 is going to the counter (round 4) - # of students at counter: 4 and counter
633:Student 4 is going to the counter (round 1) - # of students at counter: 2 and counter
650:Student 6 is going to the counter (round 1) - # of students at counter: 1 and counter
652:Student 4 is going to the counter (round 2) - # of students at counter: 1 and counter
658:Student 6 is going to the counter (round 2) - # of students at counter: 2 and counter
671:Student 6 is going to the counter (round 3) - # of students at counter: 2 and counter
678:Student 4 is going to the counter (round 3) - # of students at counter: 2 and counter
680:Student 8 is going to the counter (round 1) - # of students at counter: 3 and counter
702:Student 5 is going to the counter (round 4) - # of students at counter: 2 and counter
704:Student 6 is going to the counter (round 4) - # of students at counter: 2 and counter
731:Student 4 is going to the counter (round 4) - # of students at counter: 1 and counter
744:Student 8 is going to the counter (round 2) - # of students at counter: 0 and counter
753:Student 8 is going to the counter (round 3) - # of students at counter: 0 and counter
762:Student 8 is going to the counter (round 4) - # of students at counter: 0 and counter
```

And finally, I trim them to see them better.

```
    grep -nr "# of students" test.txt | cut -d ' ' -f 1,16

193:Student 0
211:Student 0
226:Student 0
236:Student 0
242:Student 0
258:Student 0
305:Student 0
312:Student 1
333:Student 1
346:Student 1
359:Student 1
389:Student 0
403:Student 0
412:Student 0
432:Student 0
442:Student 1
443:Student 2
444:Student 2
445:Student 3
446:Student 2
467:Student 4
469:Student 1
475:Student 5
477:Student 1
485:Student 6
494:Student 6
503:Student 6
525:Student 5
531:Student 5
543:Student 5
552:Student 5
566:Student 4
578:Student 4
583:Student 4
597:Student 4
604:Student 4
615:Student 4
633:Student 2
650:Student 1
652:Student 1
658:Student 2
671:Student 2
678:Student 2
680:Student 3
702:Student 2
704:Student 2
731:Student 1
744:Student 0
753:Student 0
762:Student 0
```

This is small script to get bulk data of the number of students waiting at the counter, the number of plates at the counter and the number of plates at the kitchen across time.

```
#!/bin/sh
grep -nr "# of students" test.txt | cut -d ' ' -f 1 > x1.txt
grep -nr "# of students" test.txt | cut -d ' ' -f 16,17 > y1.txt
grep -nr "counter items" test.txt | cut -d ' ' -f 1 > x2.txt
grep -nr "counter items" test.txt | cut -d ' ' -f 20 > y2.txt
grep -nr "The supplier delivered" test.txt | cut -d ' ' -f 1 > x3.txt
grep -nr "The supplier delivered" test.txt | cut -d ' ' -f 10,11 > y3.txt
```

And I have written a small script to get the data quickly from the generated text files. Our job has become very easy this way. Thanks to this python and shell script, we will collect the data just by playing the parameters and then we will examine them all.

```
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure

from datetime import datetime
import os


def get_x(file):
    with open(file, 'r') as file:
        data = file.read().replace(':', ' ')
    return [int(s) for s in data.split() if s.isdigit()]

def get_y1(file):
    with open(file, 'r') as file:
        data = file.read()
    return [int(s) for s in data.split() if s.isdigit()]

def get_y(file):
    with open(file, 'r') as file:
        data = file.read().replace('=', ' ')
    return [int(s) for s in data.split() if s.isdigit()]

figure(num=None, figsize=(10, 10), dpi=120, facecolor='w', edgecolor='k')


os.system('./info.sh')


x1 = get_x("../source/x1.txt")
y1 = get_y1("../source/y1.txt")
x2 = x1
y2 = get_y("../source/y2.txt")
x3 = get_x("../source/x3.txt")
y3 = get_y("../source/y3.txt")


plt.subplot(3, 1, 1)
plt.plot(x1, y1, 'ro')
plt.ylabel('# of students at counter')

plt.subplot(3, 1, 2)
plt.plot(x2, y2, 'bo')
plt.ylabel('# of plates at counter')

plt.subplot(3, 1, 3)
plt.plot(x3, y3, 'go')
plt.ylabel('# of plates at kitchen')
plt.xlabel('time')


# plt.axis([0, 800, 0, 10])
now = datetime.now()

current_time = now.strftime("%H:%M:%S")

plt.savefig('plots/graph{0}.png'.format(current_time))
```

Now that we have shown the strategy, we can now show the graphics and start to analysis.

3.1 Cooks (N)

```
./program -N [3, 8] -T 5 -S 4 -L 5 -U 8 -G 2 -F supply.txt
```

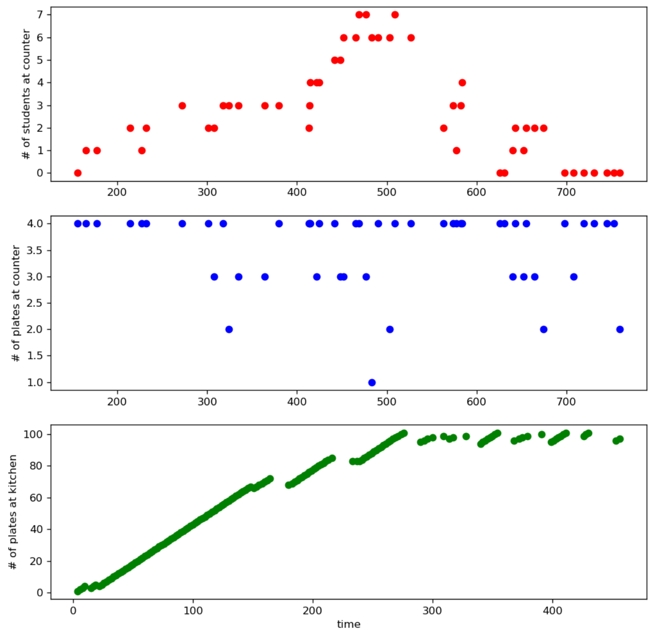


Figure 3: N = 3

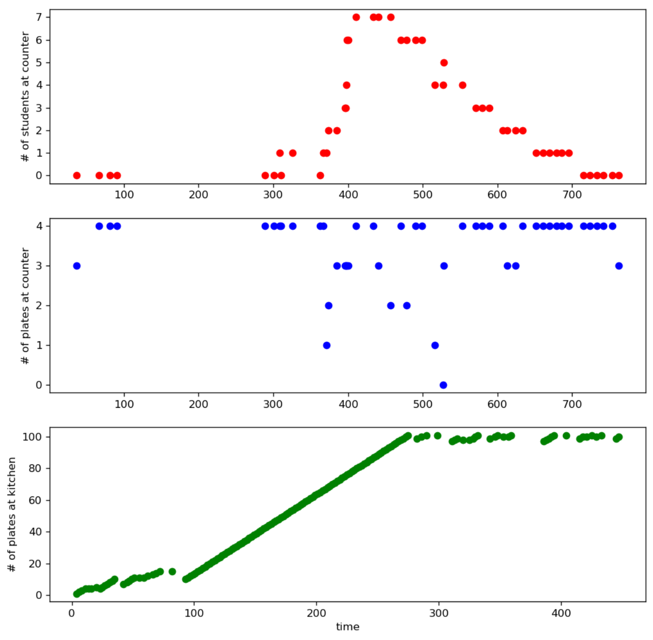


Figure 4: N = 4

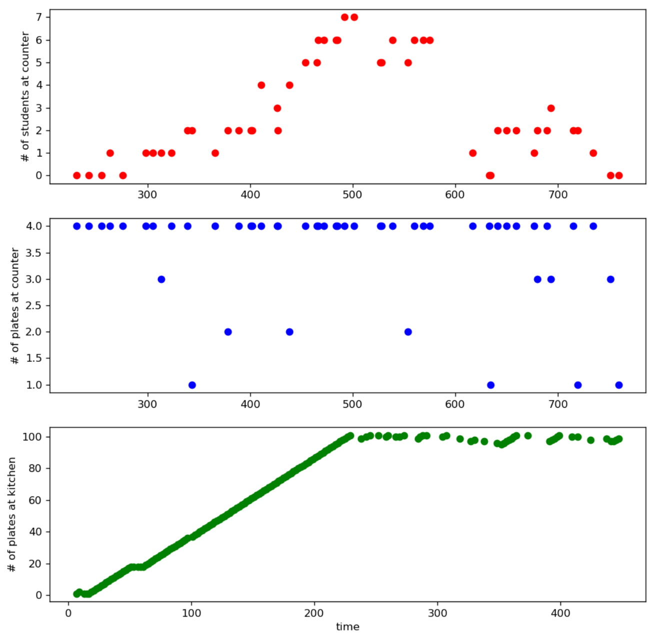


Figure 5: N = 5

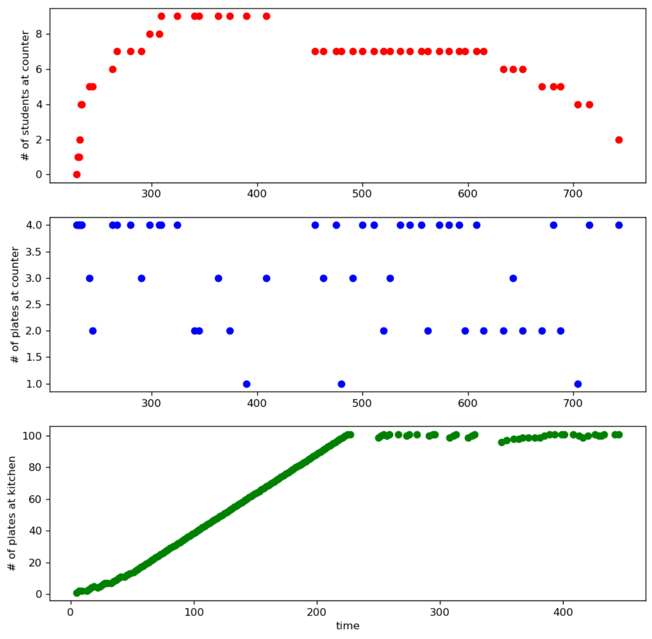


Figure 6: N = 6

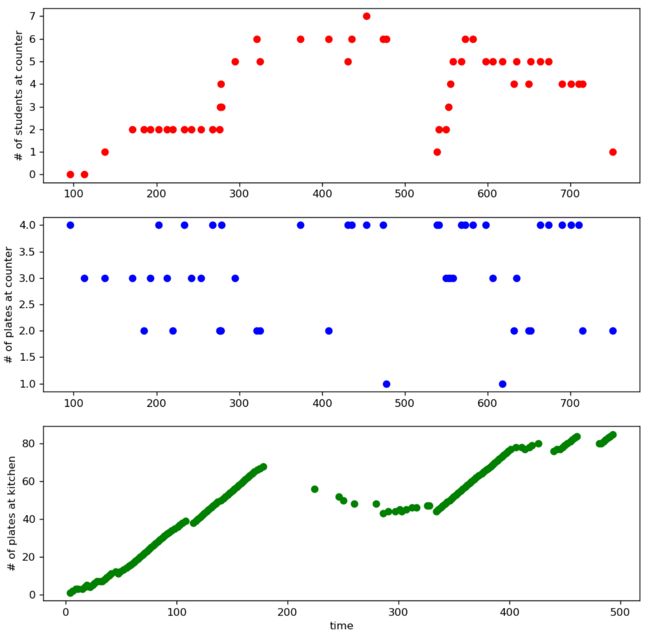


Figure 7: N = 7

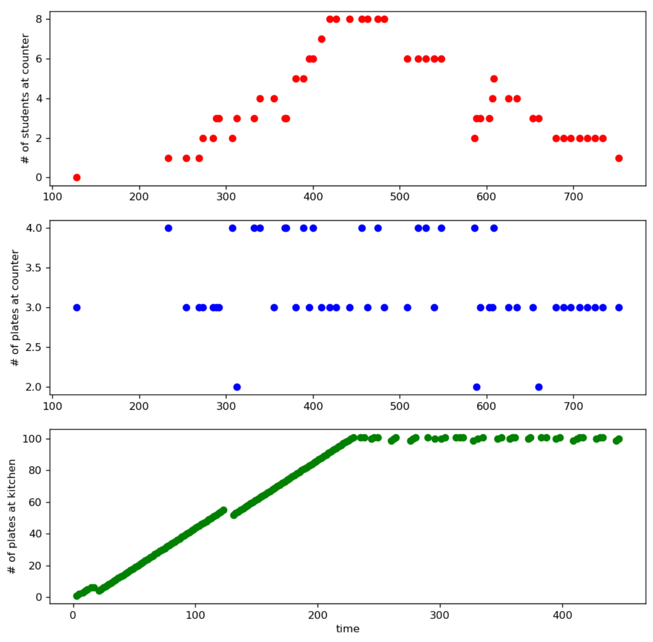


Figure 8: N = 8

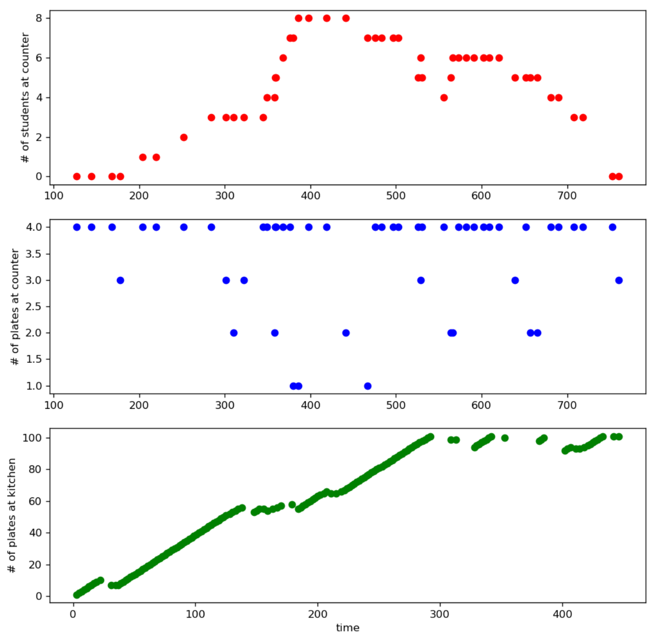


Figure 9: N = 9

3.2 Tables (T)

```
./program -N 4 -T [3, 8] -S 4 -L 5 -U 8 -G 2 -F supply.txt
```

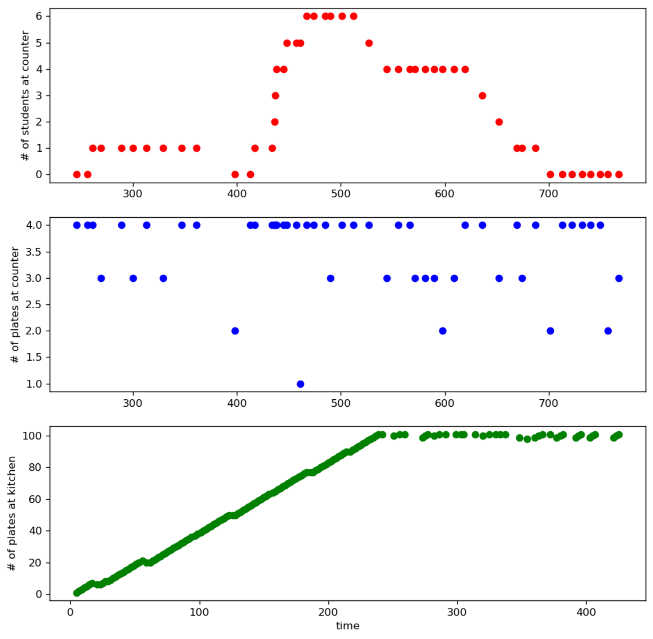


Figure 3: T = 3

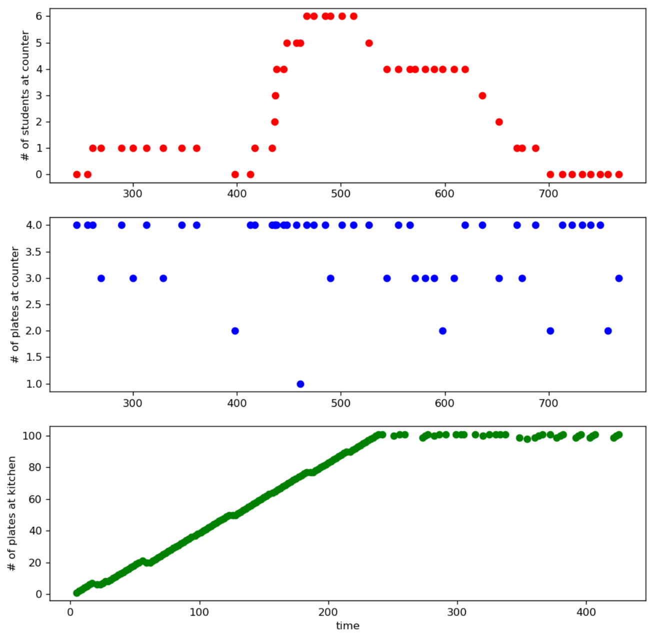


Figure 4: T = 3

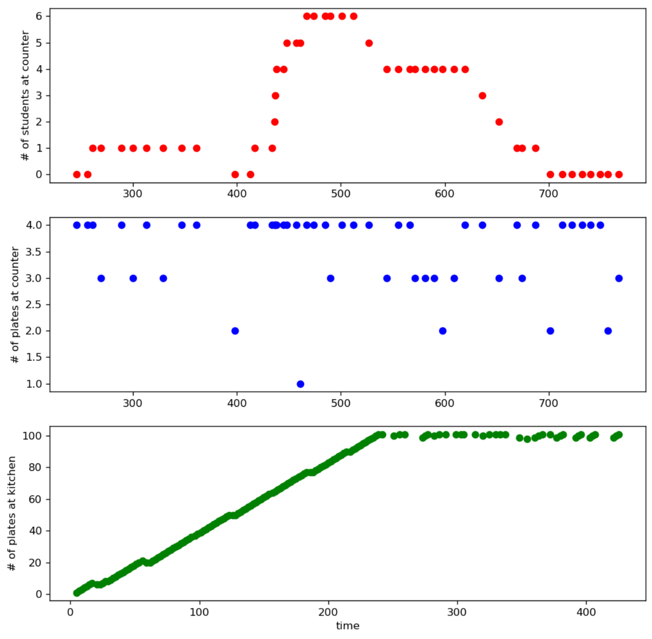
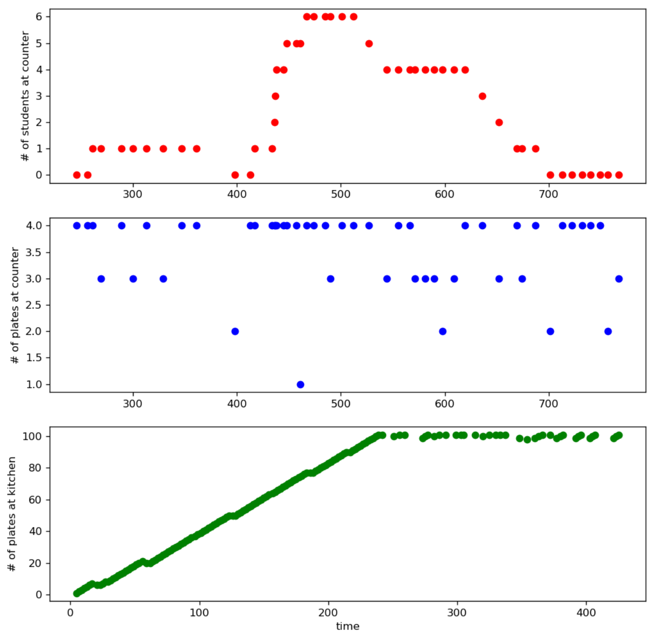


Figure 5: T = 3



# of students at counter
# of plates at counter
# of plates at kitchen
time

Figure 6: T = 3

3.3 Counter Size (S)

```
./program -N 4 -T 5 -S [4, 10] -L 5 -U 8 -G 2 -F supply.txt
```

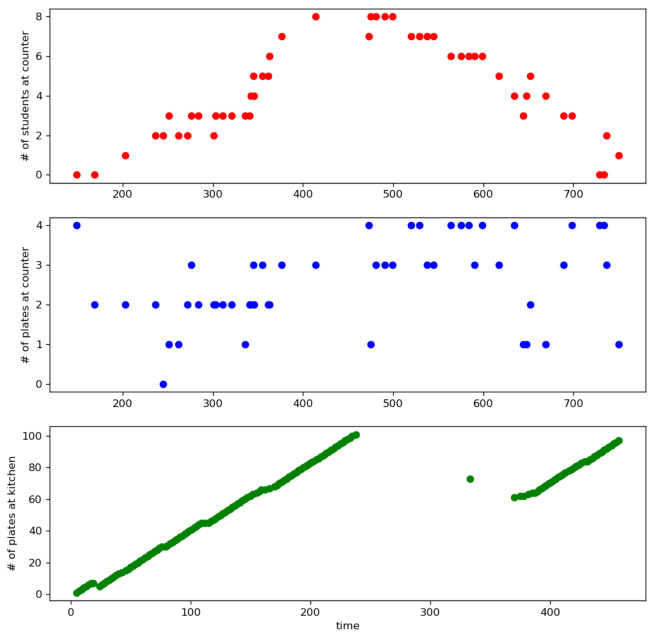


Figure 3: S = 4

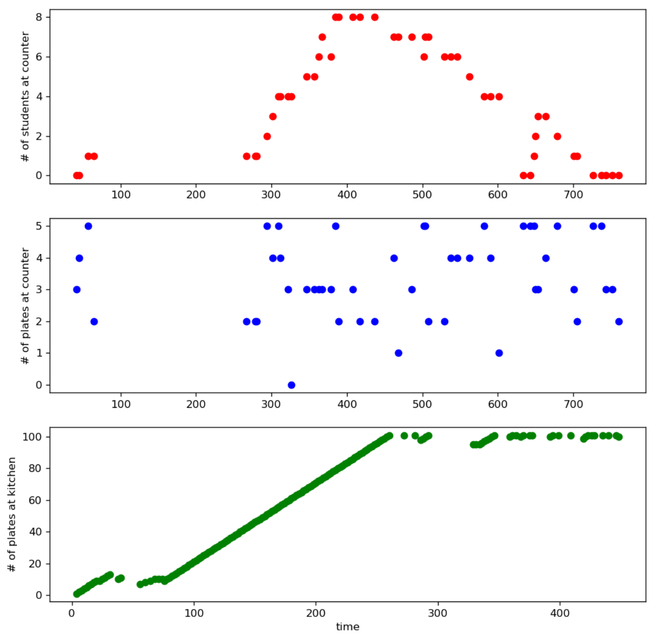


Figure 4: S = 5

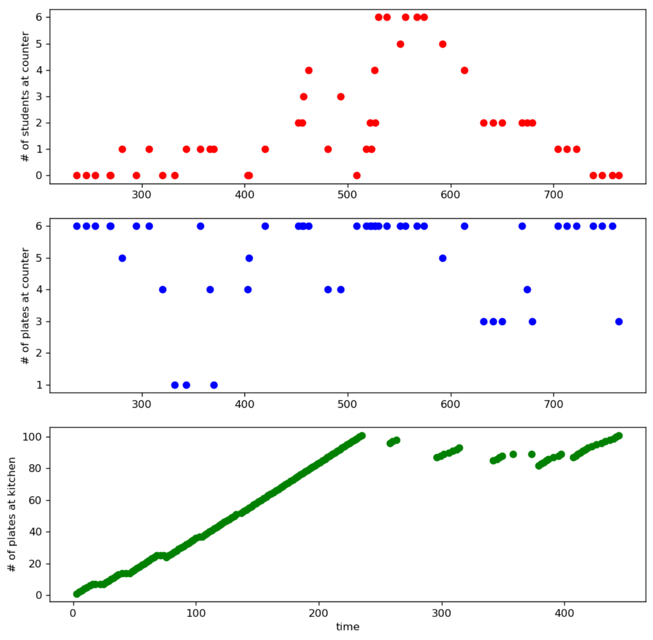


Figure 5: S = 6

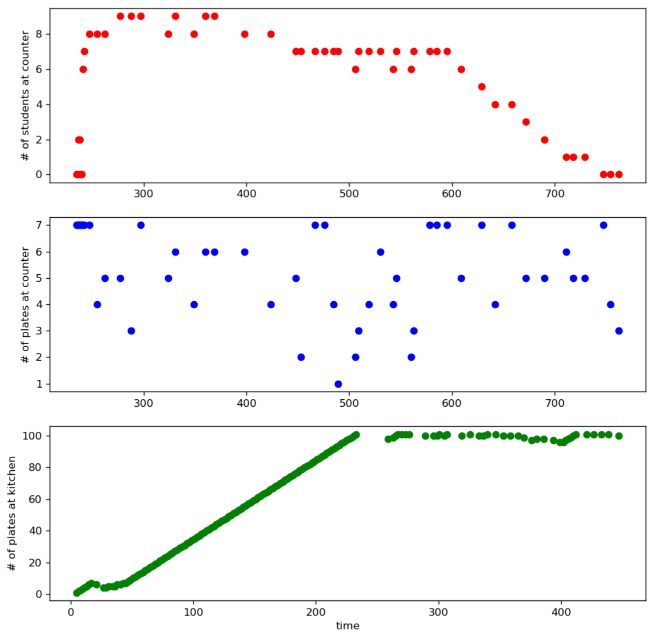


Figure 6: S = 7

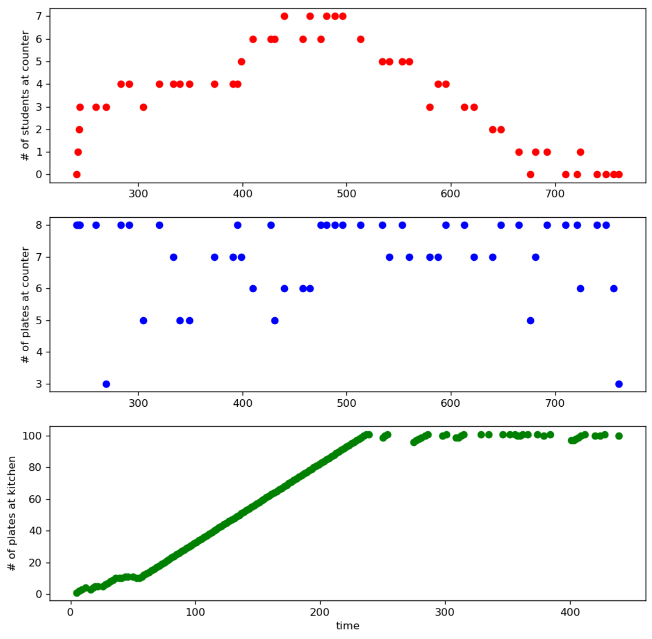


Figure 7: S = 8

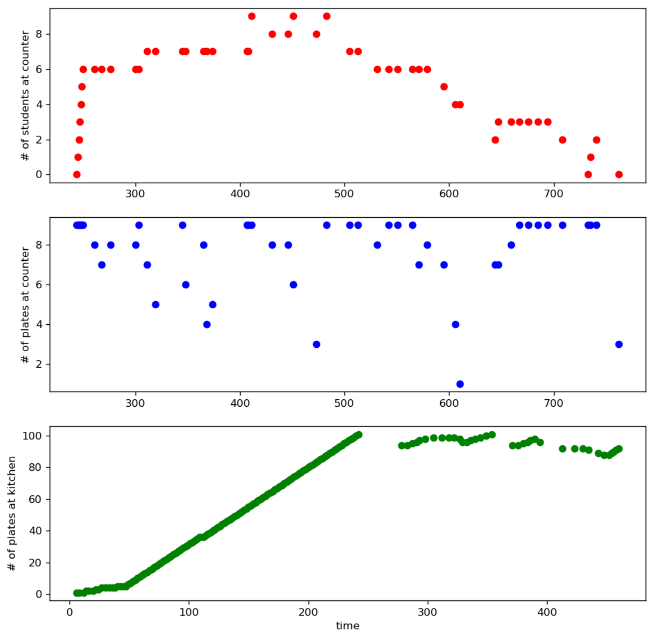


Figure 8: S = 9

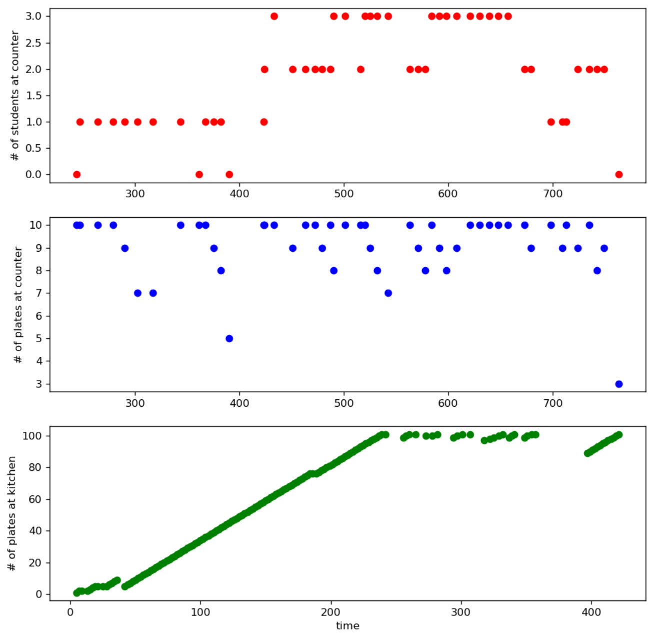


Figure 9: S = 10

3.4 Round Size (L)

```
./program -N 4 -T 4 -S 5 -L [5, 10] -U 8 -G 2 -F supply.txt
```

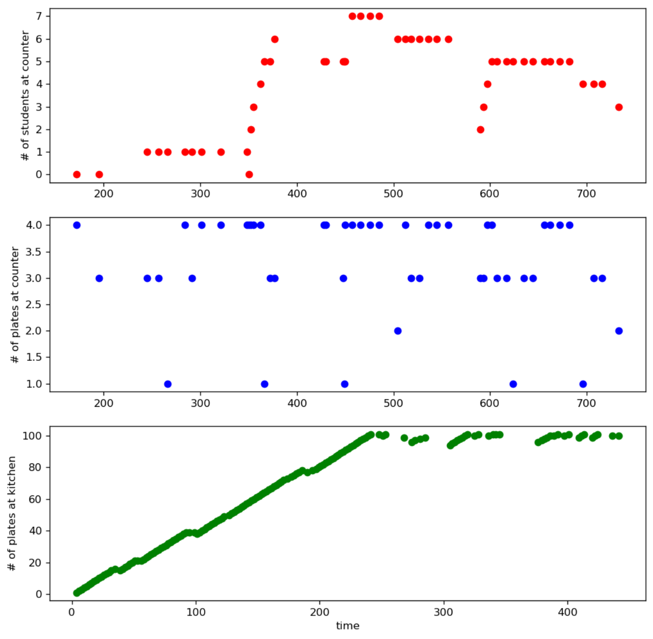


Figure 3: L = 5

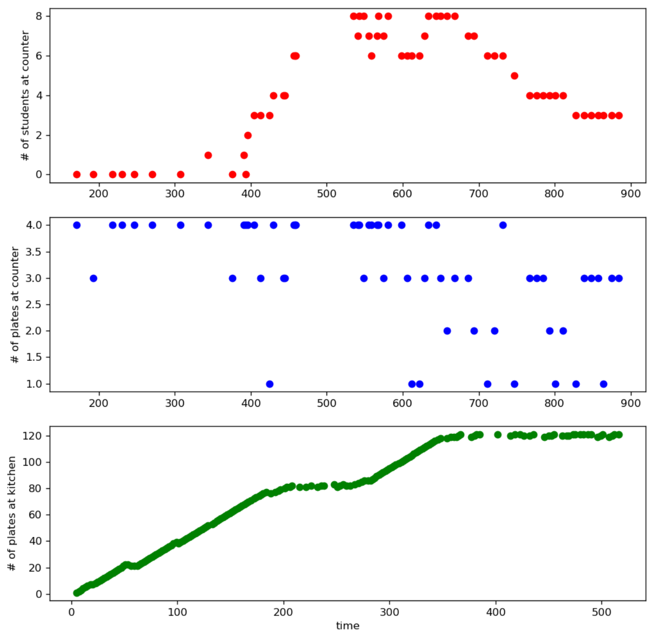


Figure 4: L = 6

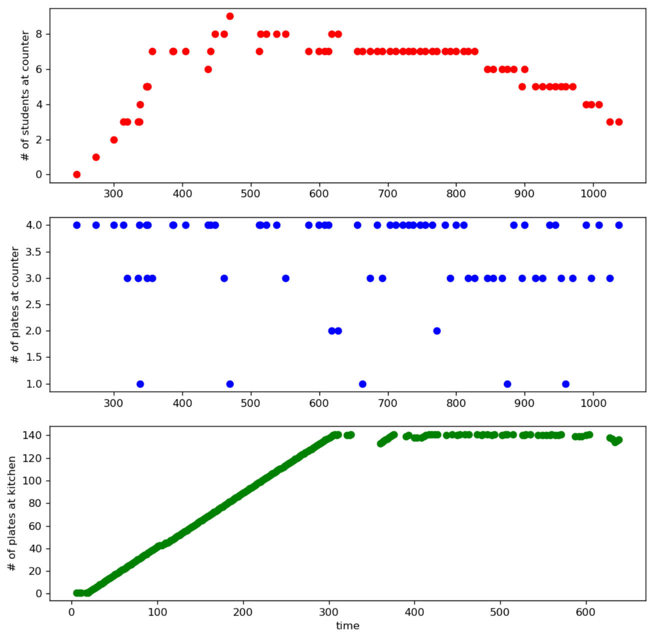


Figure 5: L = 7

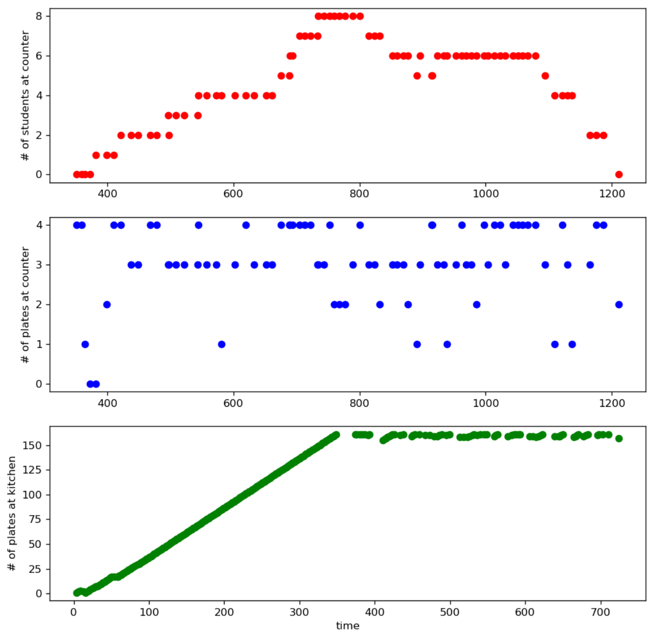


Figure 6: L = 8

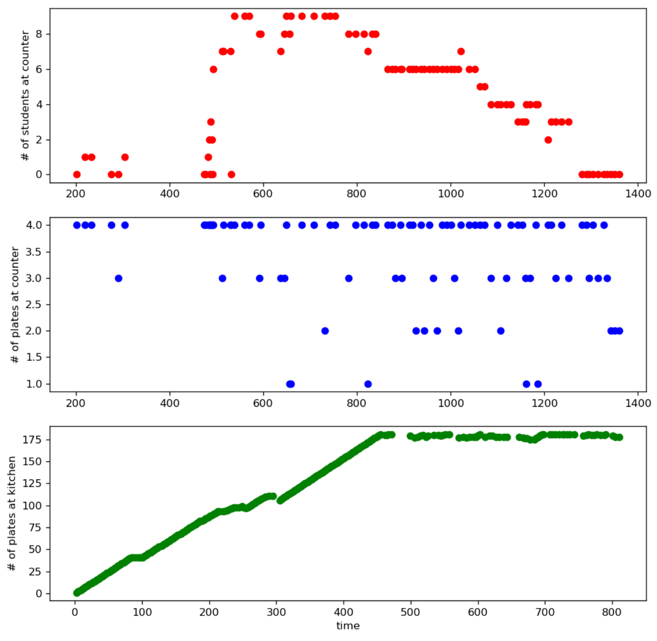


Figure 7: L = 9

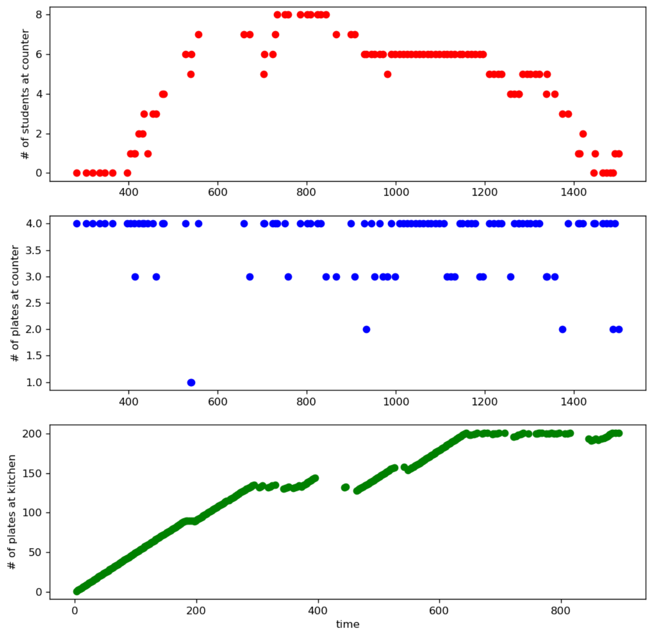

# of students at counter
# of plates at counter
# of plates at kitchen
time

Figure 8: L = 10

3.5 Undergraduate Student (U)

```
./program -N 4 -T 4 -S 5 -L 5 -U [8, 13] -G 2 -F supply.txt
```

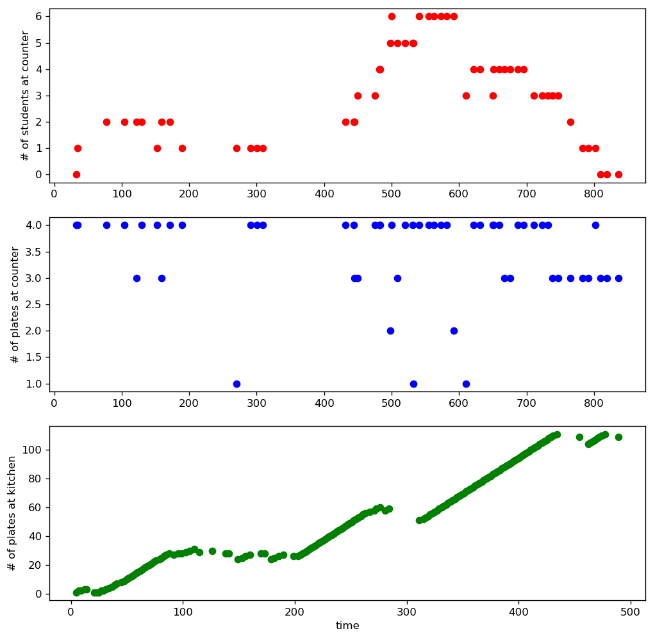


Figure 3: U = 8

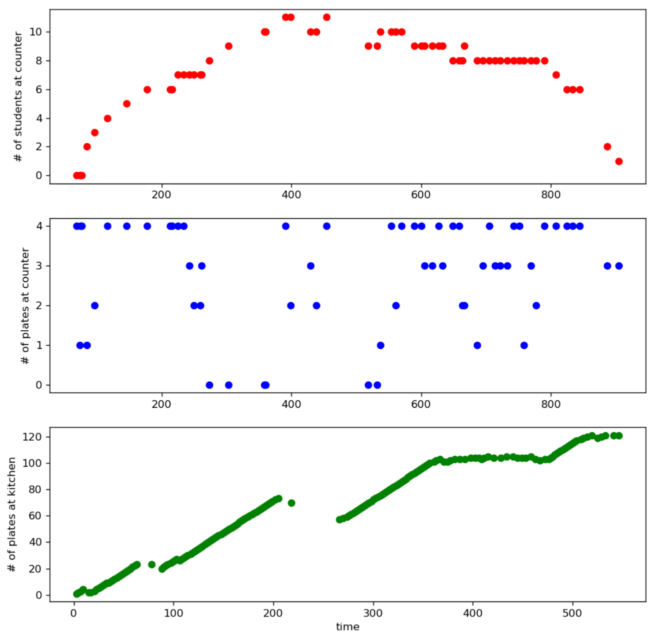


Figure 4: U = 9

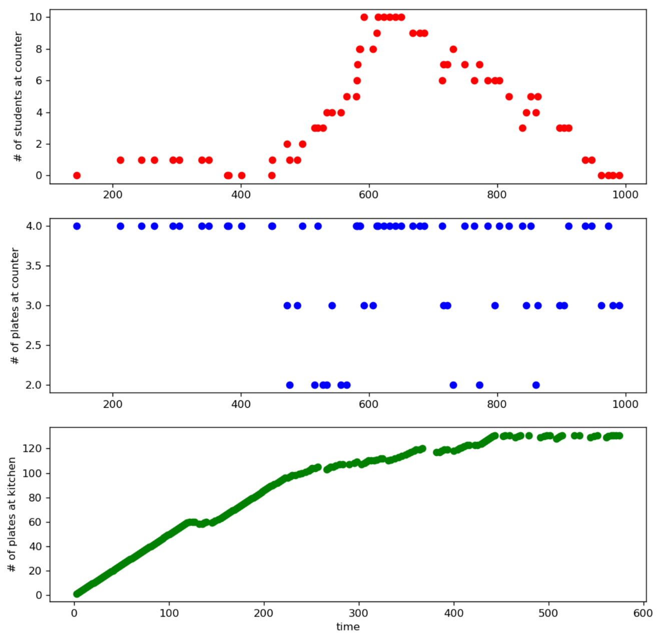


Figure 5: U = 10

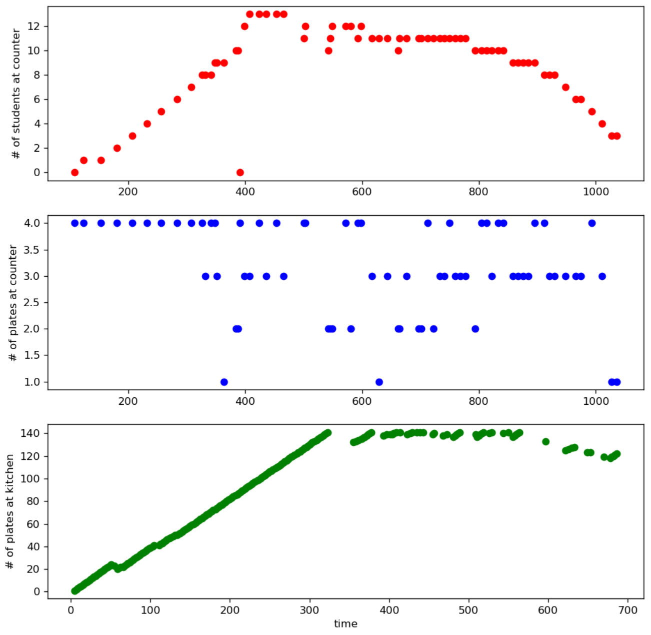


Figure 6: U = 11

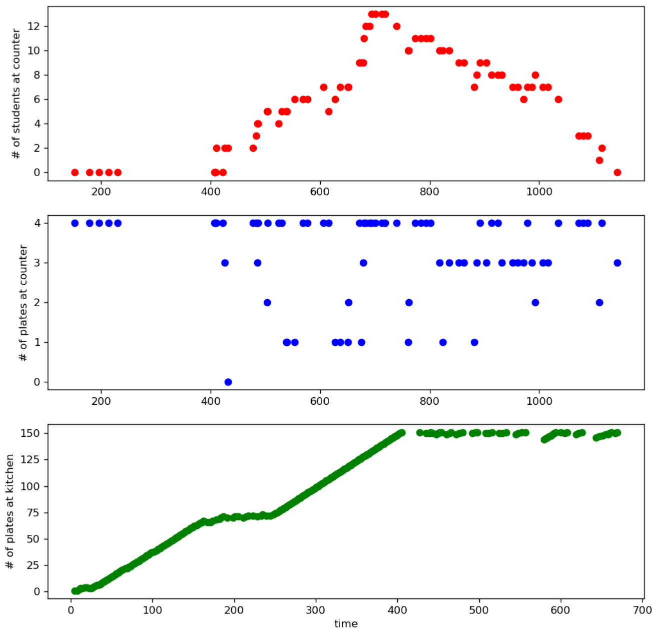
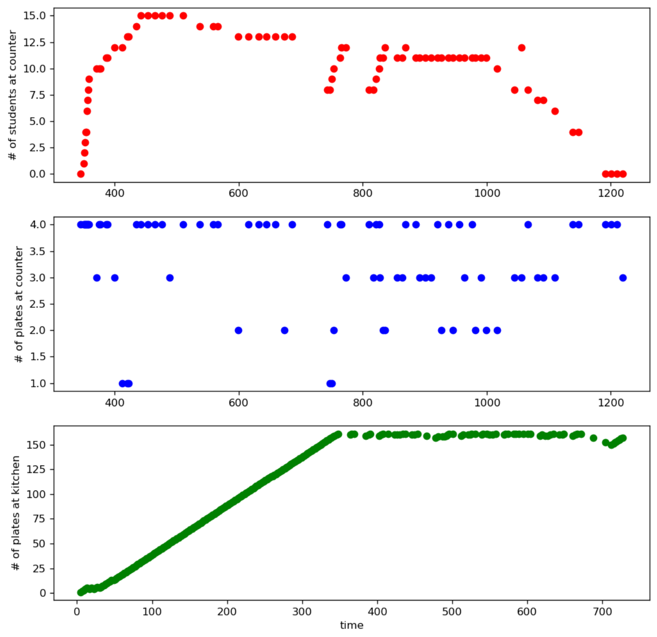


Figure 7: U = 12



# of students at counter
# of plates at counter
# of plates at kitchen
time

Figure 8: U = 13

3.6 Graduate Student (G)

```
./program −N 4 −T 4 −S 5 −L 5 −U 8 −G [2, 9] −F supply.txt
```

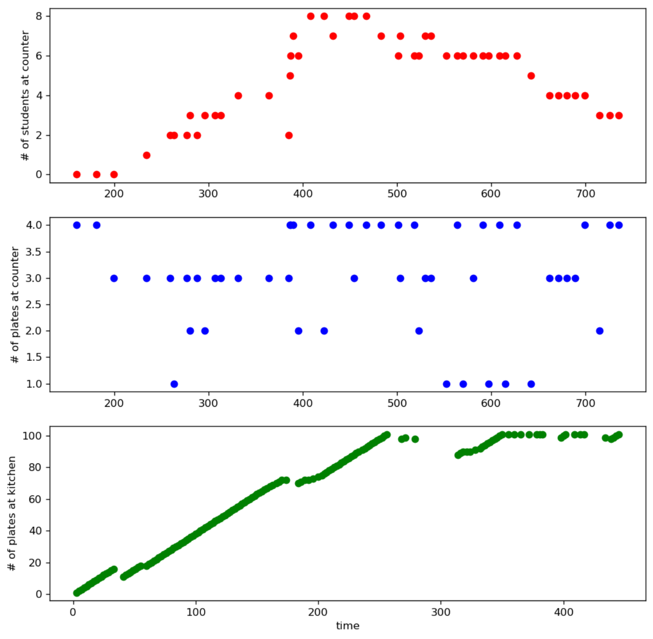


Figure 3: G = 2

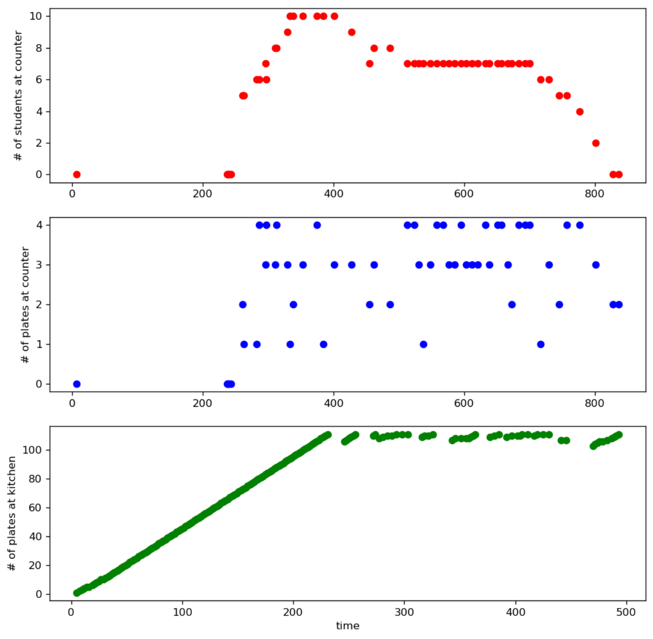


Figure 4: G = 3

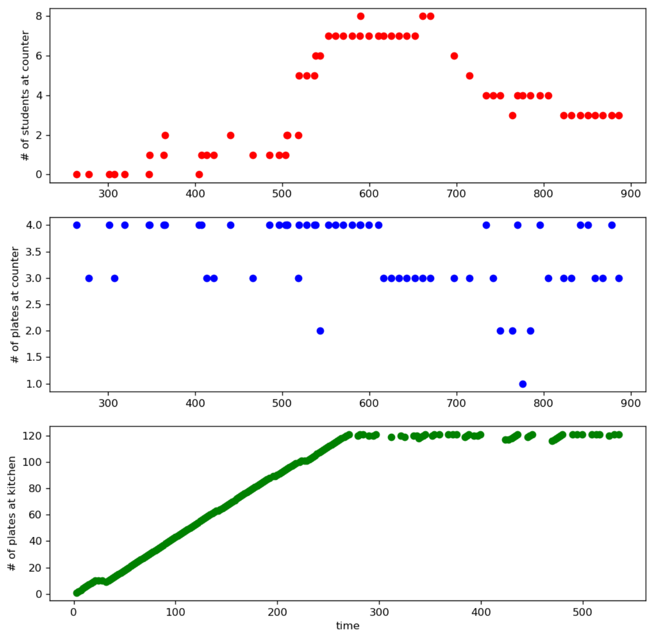


Figure 5: G = 4

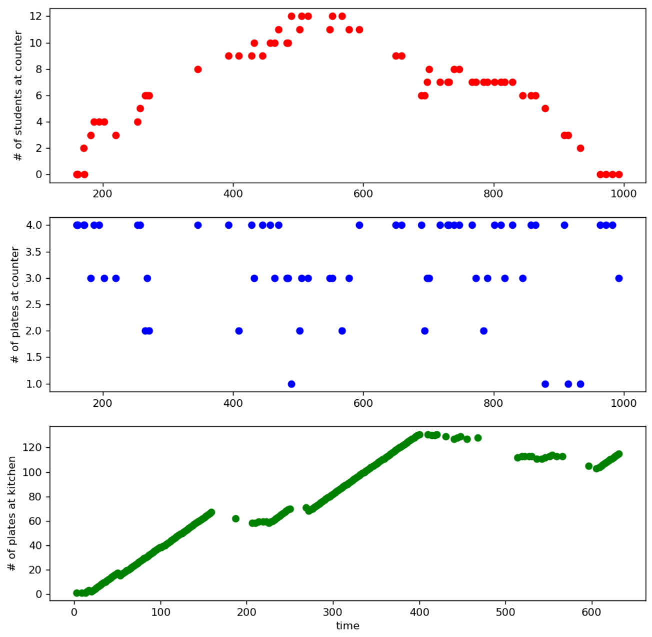


Figure 6: U = 11

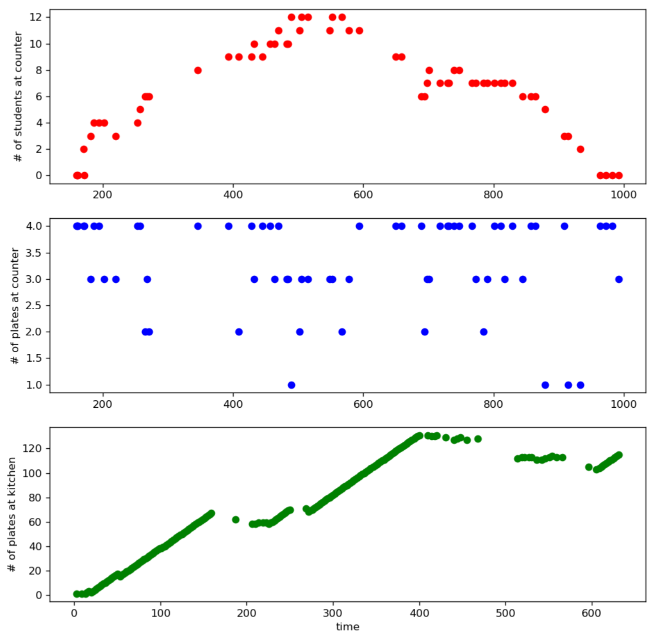


Figure 7: G = 5

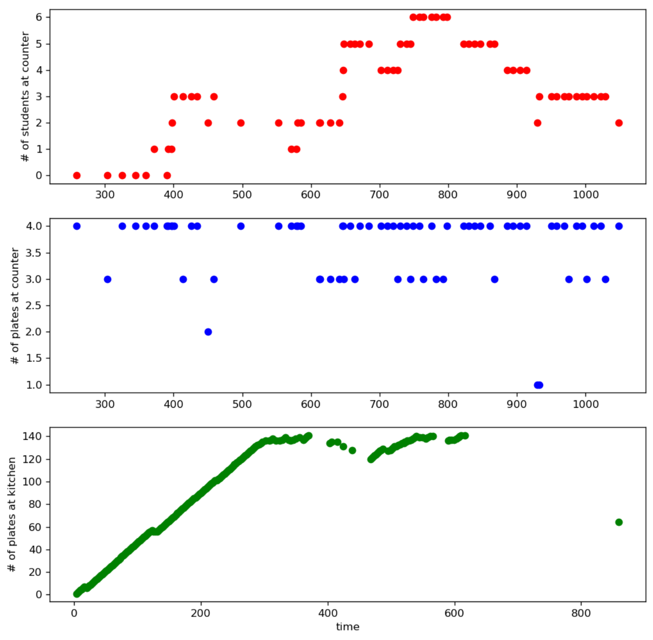


Figure 8: G = 6

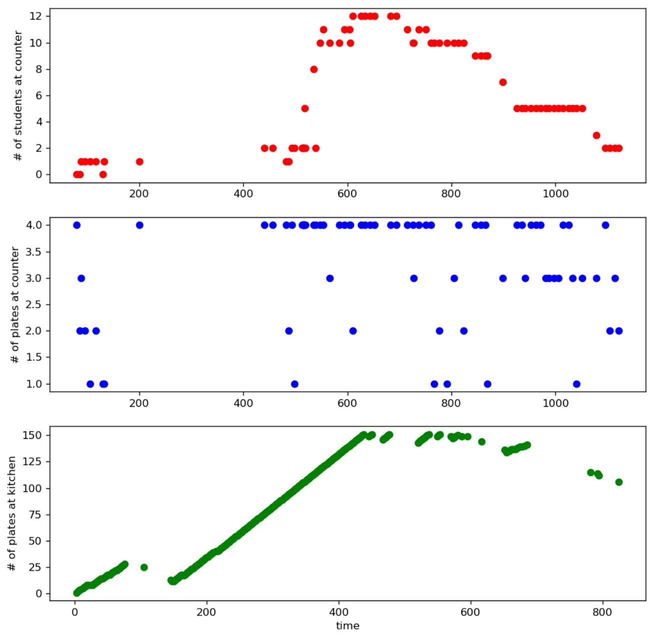


Figure 9: G = 7