- The details behind your implementation. As part of this discussion write answers to the following questions:
    - How does your solution guarantee that only one process will attempt to remove a carrot at a given time?

      I use a semaphore for the farmers, and one semaphore for the vegetarians. I set the farmers semaphore to be 200, and vegetarians semaphore to be 0. And two functions separately for farmers and vegetarians named prod and cons. Prod function every time wait the semaphore of farmers and signal the semaphore of vegetarians, so vegetarians will know there are more available carrots there. Then cons would wait for semaphore of vegetarians and signal farmers.

      I also added a semaphore for the vegetarians, which helps to guarantee the vegetarians will use the buffer for buying carrots at the same time. I set the value to be 1 at first, and when it gets into one of the process it wait the semaphore and when it had done with the push, it signal this semaphore.

    - In your solution, when a producer needs to insert a carrot in the queue, are consumers also excluded, or can consumers continue concurrently?

      The consumers are excluded. Since I think the buffer to store the carrots can be access only by consumers or producers, otherwise there would be conflict on the processor.

    - When a consumer starts to extract carrots, does your system guarantee that the consumer will receive contiguous locations in the queue, or do consumers contend for carrots?

      Yes, the consumer will receive contiguous locations, if the hunger is 2, he will have two contiguous carrots. He will wait when he get the first one until he gets the second.

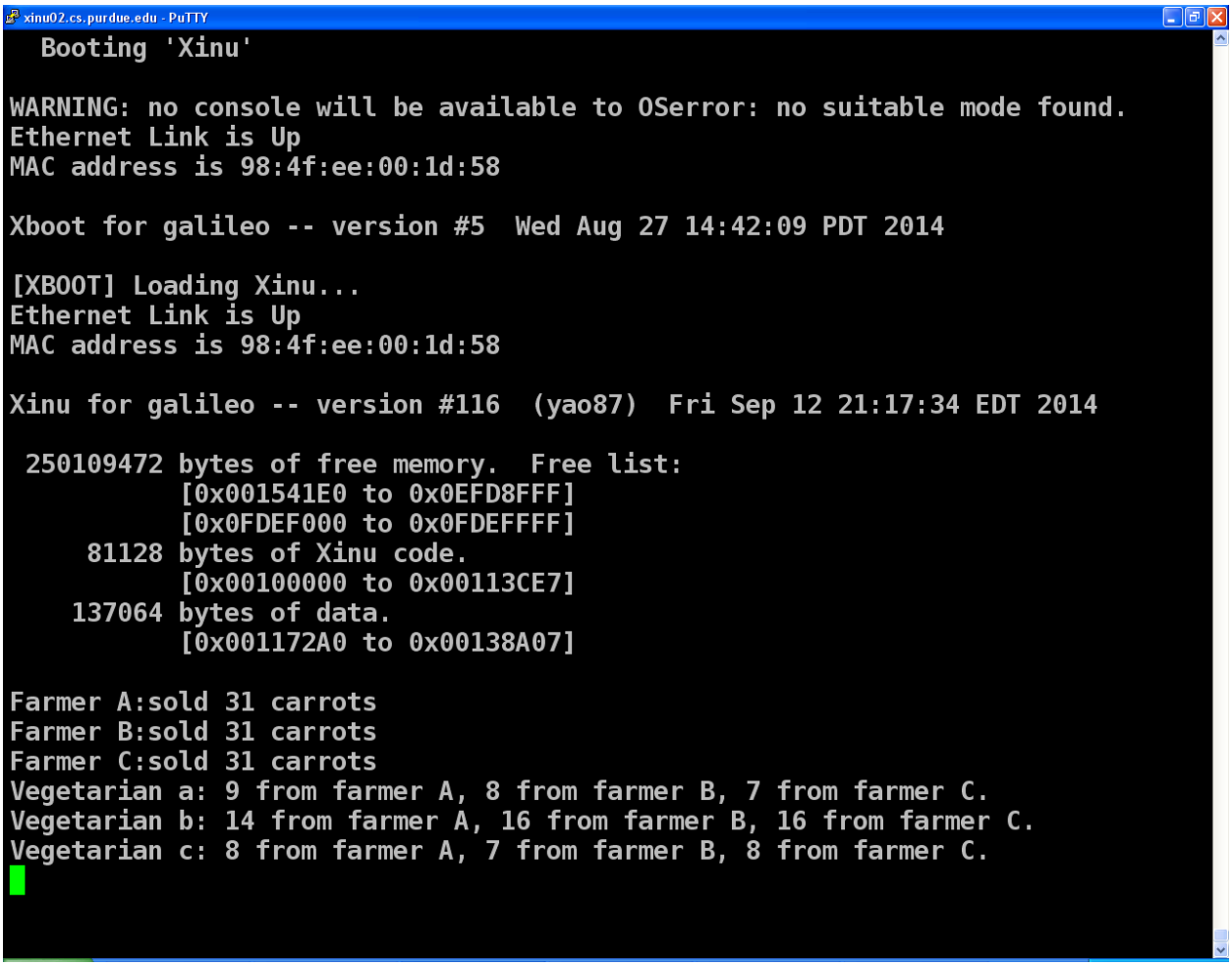
- The steps you took to complete the requirements
    1. I used three semaphores totally. One for the control of the vegetarians competing for the carrots, one for the control of making sure the vegetarians and farmers do not use the buffer at

the same time. The semaphore of farmers is set to be 200, and semaphore of vegetarians set to be 0. Another semaphore which controls the vegetarians competing for carrots is also set to be 0.

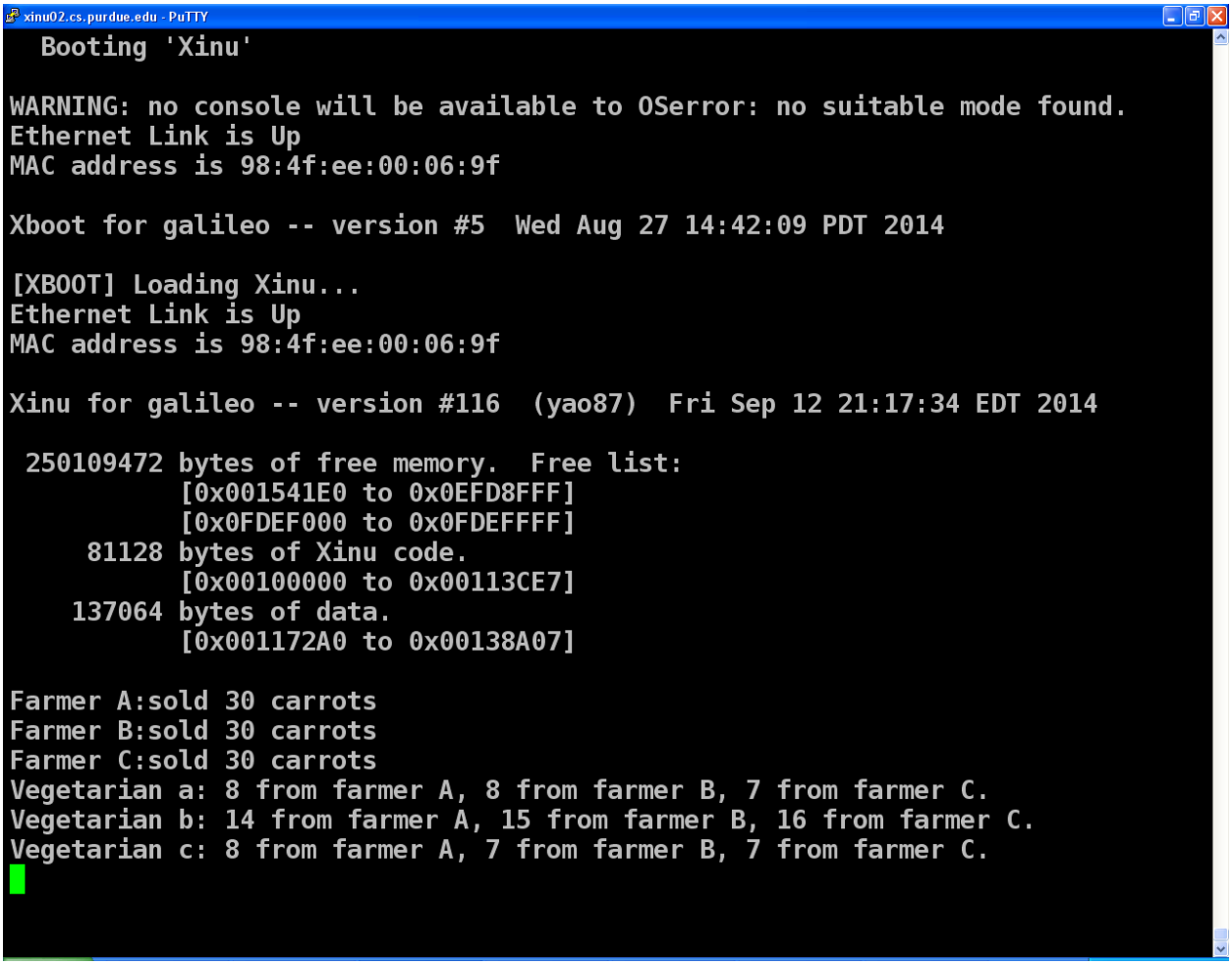
2. Then I create NVEGETARIANS + NFARMERS processes, in the example case it's 6 processes. Farmers use the function called prod- production process, and Vegetarians use the function called cons- consumption. Then I resume all the processes.
3. I used a buffer of circular queue to restore the field carrots.
4. The prod function first waits for semaphore of farmers, since it's 200, it will produce first until this semaphore goes down to 0 (usually it wouldn't). then it push the tags of the farmers into the buffer, so when we pop it out we will know who it belongs to. Then we signal the semaphore of the vegetarians and sleep.
5. When the process of vegetarian has been signaled, it will go to the ready list. The prod function first wait for the semaphore of vegetarians which controls the vegetarians competing for carrots and then goes into the loop of hungers and wait for the semaphore of interaction with farmers. Then it pop the result from the queue and restore the result into an array. Finally it signal these two semaphore mentioned above and sleep.
6. Finally, I delete all the semaphores and kill all the processes.

- Your results from the extra credit experiment

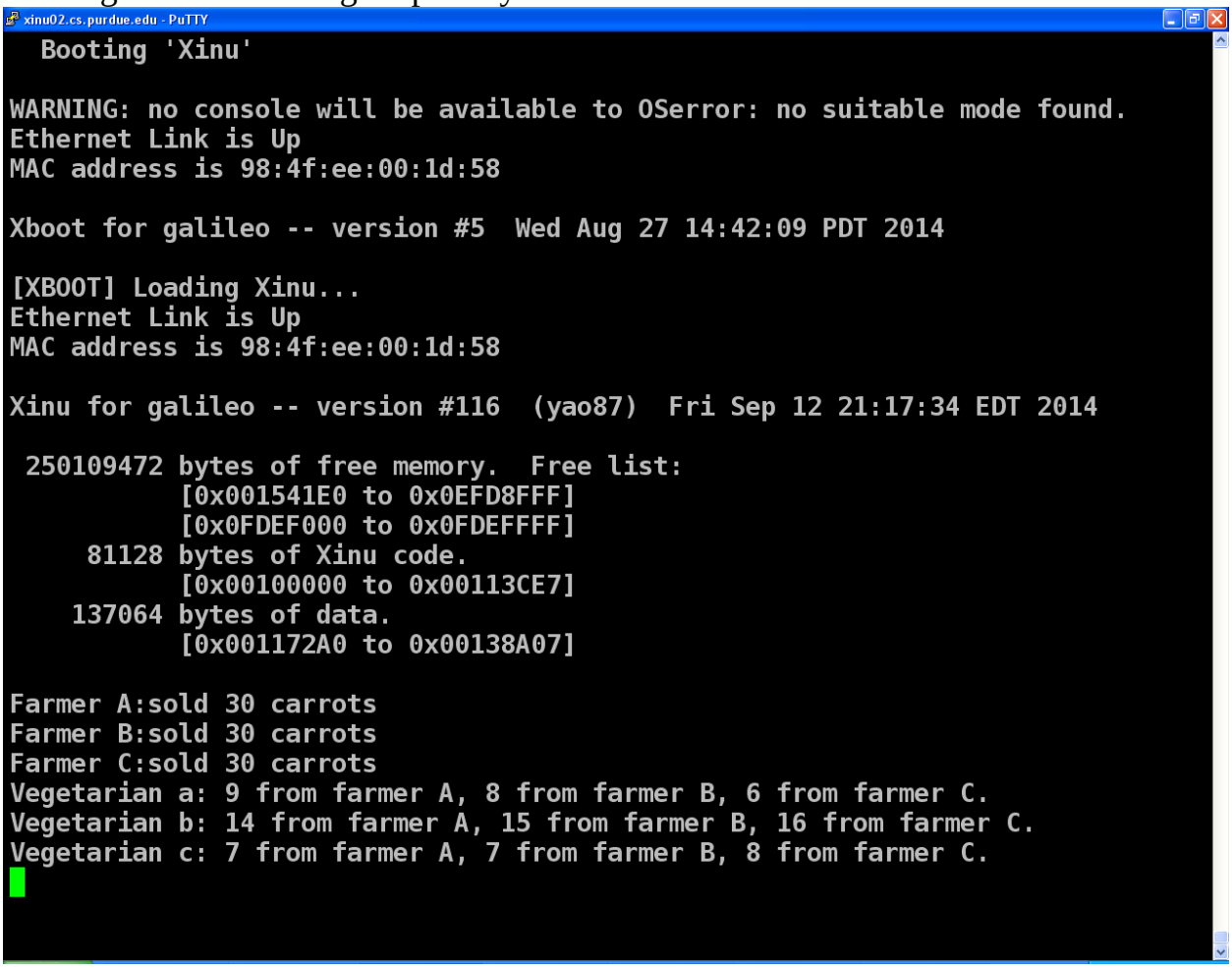
0. With the same priority:


xinu02.cs.purdue.edu - PuTTY
Booting 'Xinu'
WARNING: no console will be available to OSerror: no suitable mode found.
Ethernet Link is Up
MAC address is 98:4f:ee:00:1d:58
Xboot for galileo -- version #5 Wed Aug 27 14:42:09 PDT 2014
[XBOOT] Loading Xinu...
Ethernet Link is Up
MAC address is 98:4f:ee:00:1d:58
Xinu for galileo -- version #116 (yao87) Fri Sep 12 21:17:34 EDT 2014
250109472 bytes of free memory. Free list:
[0x001541E0 to 0x0EFD8FFF]
[0x0FDEF000 to 0x0FDEFFFF]
81128 bytes of Xinu code.
[0x00100000 to 0x00113CE7]
137064 bytes of data.
[0x001172A0 to 0x00138A07]
Farmer A:sold 31 carrots
Farmer B:sold 31 carrots
Farmer C:sold 31 carrots
Vegetarian a: 9 from farmer A, 8 from farmer B, 7 from farmer C.
Vegetarian b: 14 from farmer A, 16 from farmer B, 16 from farmer C.
Vegetarian c: 8 from farmer A, 7 from farmer B, 8 from farmer C.

1. The farmers have higher priority than the vegetarians

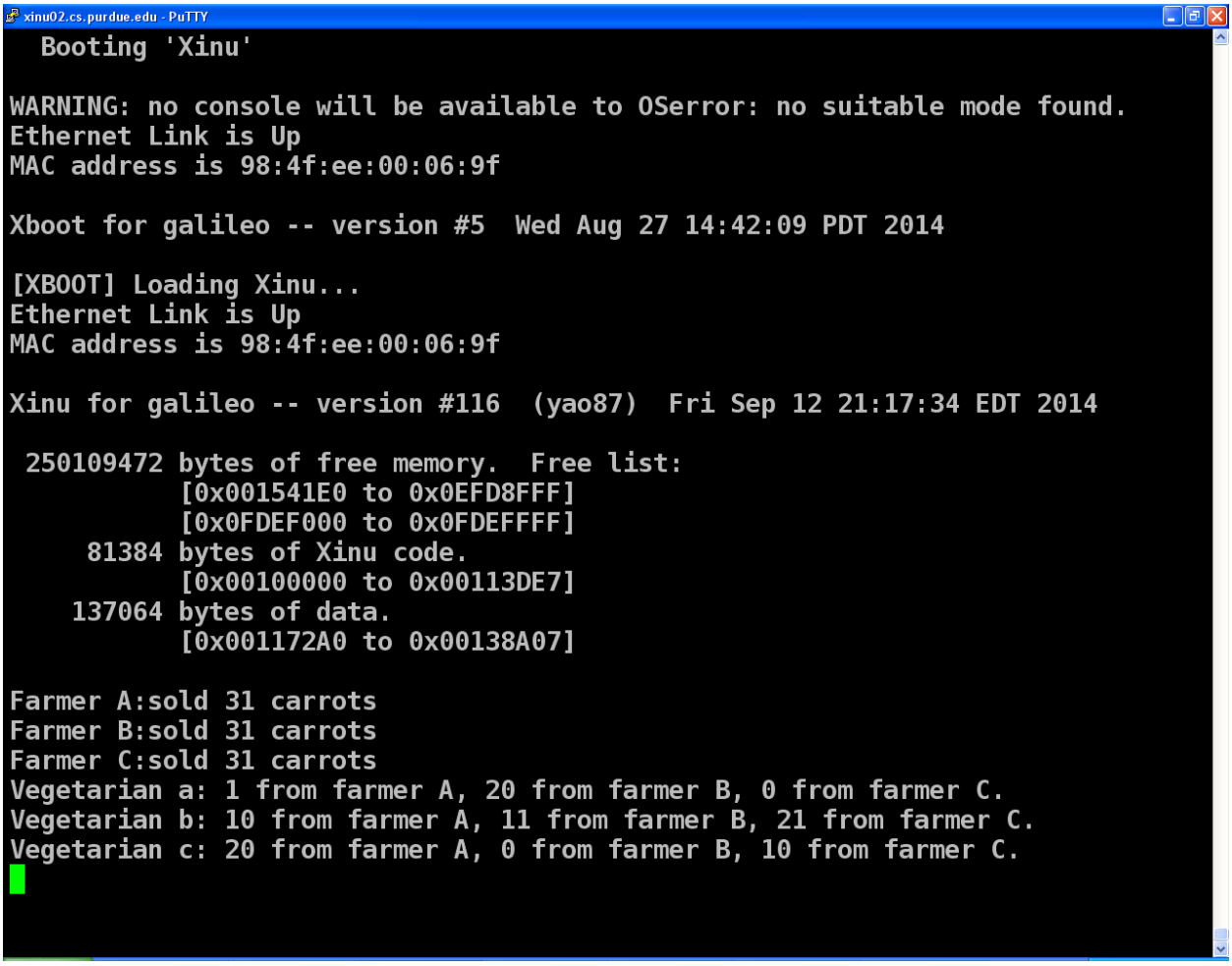


2. The vegetarians have higher priority than the farmers

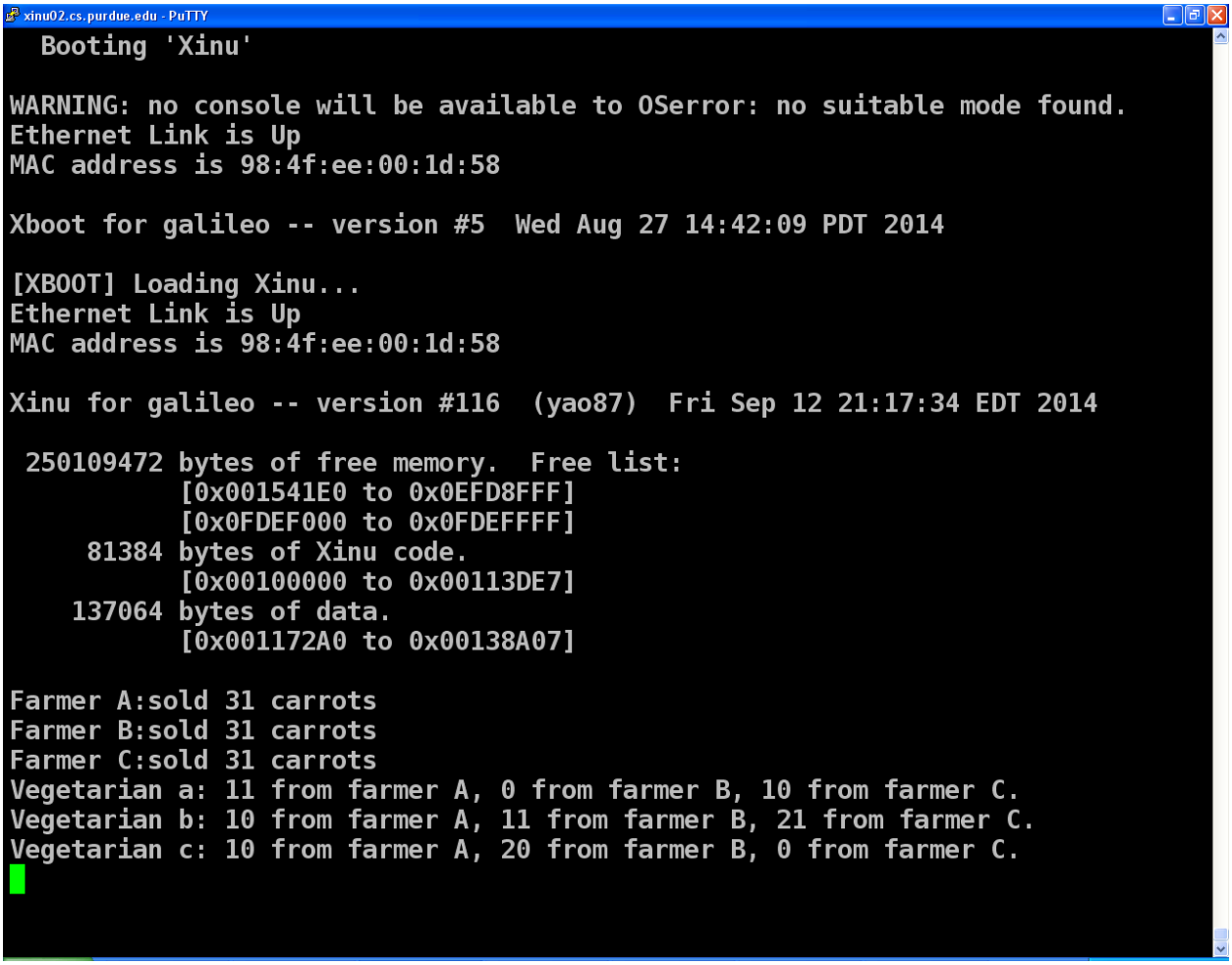


3. Each vegetarian to have a unique priority (high, medium, low)

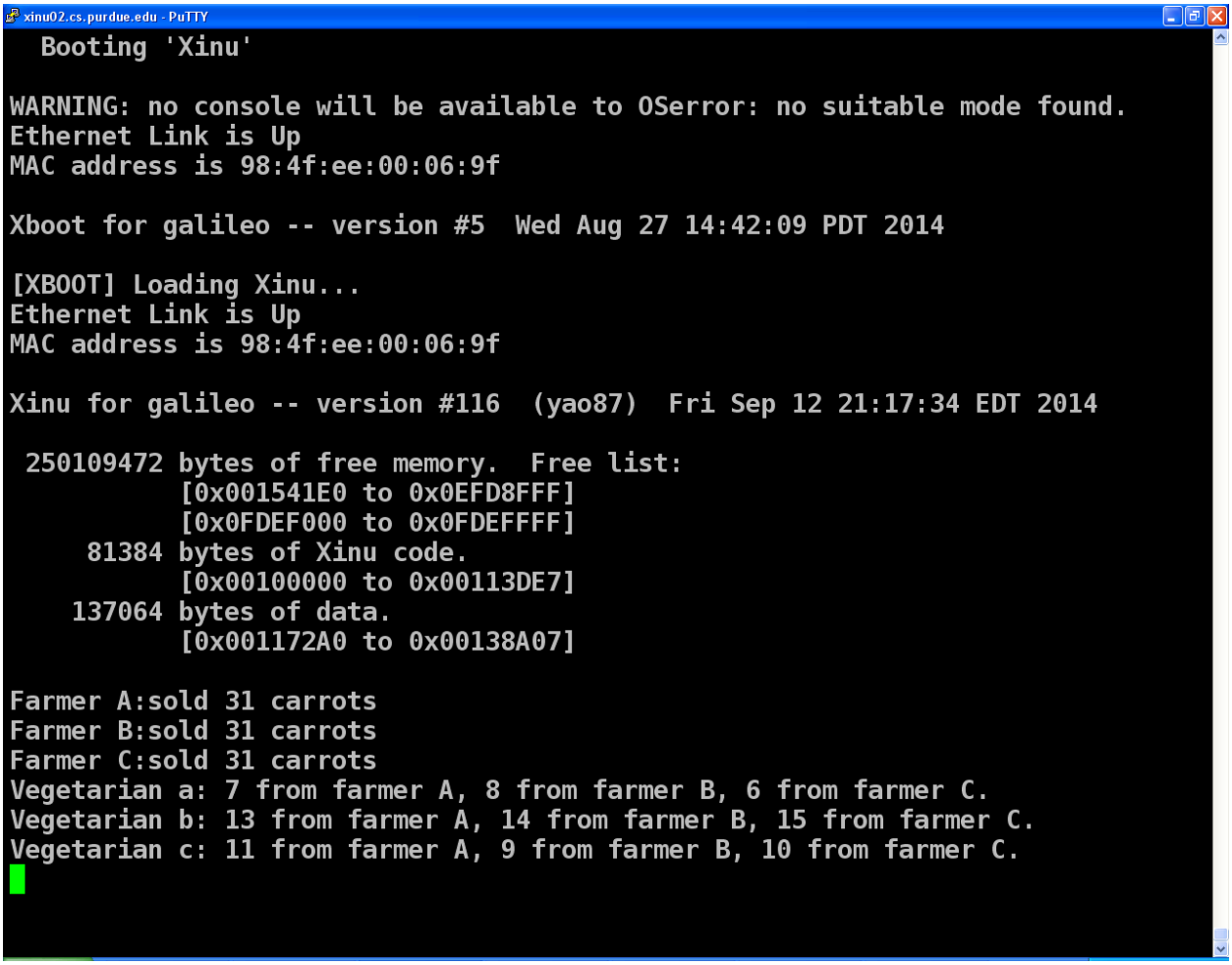
Farmers priority to be 200, Vegetarians priority to be 150 200 250 separately for a b c.



Farmers priority to be 200, Vegetarians priority to be 250 300 350 separately for a b c.



Farmers priority to be 200, Vegetarians priority to be 50 100 150 separately for a b c.



```
/*  farm.c  - definition of farmer and vegetarian lab */

#include <xinu.h>

#include "farm.h"

/*------------------------------------------------------------------------

 *  start_farm  -  Initializes and starts the farm simulation

 *------------------------------------------------------------------------

 */

#include <stdio.h>

#include <stdlib.h>

struct buffer {

            int size;

   int start;

   //int end;  // position of last element

   /* Tracking start and end of buffer would waste

    * one position. A full buffer would always have

    * to leave last position empty or otherwise

    * it would look empty. Instead this buffer uses

    * count to track if buffer is empty or full

    */

   int count; // number of elements in buffer

   /* Two ways to make buffer element type opaque

    * First is by using typedef for the element

    * pointer. Second is by using void pointer.

    */

   /* different types of buffer:
```

```
   int *element;   // array of integers
   char *element;  // array of characters
   void *element;  // array of void type (could cast to int, char, etc)
   char **element; //array of char pointers (array of strings)
   void **element; // array of void pointers
   Choosing array of void pointers since it's the most flexible */
   char element[FIELDSIZE];
};
sid32 prod_sema,cons_sema,cons_sema_t;
typedef struct buffer buffer_t;
sid32 farmer[NFARMERS];
pid32 prc_farmer[NFARMERS];
sid32 vegetarian[NVEGETARIANS];
pid32 prc_vege[NVEGETARIANS];
char result[NVEGETARIANS][FIELDSIZE*NFARMERS];
void init_buffer(buffer_t *buffer, int size) {
   buffer->size = size;
   buffer->start = 0;
   buffer->count = 0;
   //buffer->element = malloc(sizeof(buffer->element)*size);
   /* allocated array of void pointers. Same as below */
   //buffer->element = malloc(sizeof(void *) * size);

}
```

```
int full(buffer_t *buffer) {
   if (buffer->count == buffer->size) {
      return 1;
   } else {
      return 0;
   }
}
int size(buffer_t *buffer) {
            return buffer->count;
}
int empty(buffer_t *buffer) {
   if (buffer->count == 0) {
      return 1;
   } else {
      return 0;
   }
}


void push(buffer_t *buffer, char data) {
   int index;
   if (full(buffer)) {
      printf("Buffer overflow\n");
   } else {
      index = buffer->start + buffer->count++;
```

```
        if (index >= buffer->size) {
            index = 0;
        }
        buffer->element[index] = data;
    }
}



char popqueue(buffer_t *buffer) {
        char element;
                /* FIFO implementation */
                element = buffer->element[buffer->start];
                buffer->start++;
                buffer->count--;
                if (buffer->start == buffer->size) {
                        buffer->start = 0;
                }

                return element;
}

char popstack(buffer_t *buffer) {
        int index;
                /* LIFO implementation */
                index = buffer->start + buffer->count - 1;
```

```
		if (index >= buffer->size) {
			index = buffer->count - buffer->size - 1;
		}
		buffer->count--;
		return buffer->element[index];
}

void prod(sid32 selfsema,sid32 othersema, buffer_t *buf, char tags, int times){
	//kprintf("prod\n");
	while(1){
		wait(selfsema);
		push(buf, tags);
		signal(othersema);
		sleepms(times);

	}

}

void cons(int idx, sid32 cons_sema,sid32 selfsema, sid32 othersema, buffer_t *buf, char tags, int
hungers, int times){
	int j,count,j2;
	count = 0;
	while(1){
		wait(selfsema);
		for (j=0; j<hungers;j++){
```

```
                wait(cons_sema);


                //wait(selfsema);
                //sleepms(1);
                //kprintf("NNNNNN%d,MMMMMM%d,BBBBB%c",count,idx,temp);
                //kprintf("Num%d,Farmer%c,Vege%d\n",count,temp,idx);
                //push(buf_r[idx], temp);
                char temp;
                temp = popqueue(buf);
                result[idx][count]=temp;
                count++;
                }


        //for (j1=0; j1<((int)(cons_sema)-hungers);j1++){
        //      signal(nextsema);
        //}
        signal(selfsema);
        for (j2=0; j2<hungers;j2++){
                signal(othersema);
        }


        sleepms(times);
    }
}
```

```
void start_farm(void)
{
	//kprintf("start\n");
	buffer_t *buf=NULL;
	init_buffer(buf, FIELDSIZE*3);
	prod_sema = semcreate(FIELDSIZE);
	//kprintf("%d\n",prod_sema);
	cons_sema_t = semcreate(0);
	cons_sema = semcreate(1);
	int j1,j2;
	for(j1=0;j1<NVEGETARIANS;j1++){
		for(j2=0;j2<NFARMERS;j2++){
			result[j1][j2] = '\0';
		}
	}

	//kprintf("%d\n",cons_sema);
	//int bufsize = FIELDSIZE*3;
	int i,j,i1,i2;
	/*for(i3=0;i3<NVEGETARIANS;i3++){
		init_buffer(buf_r[i3], bufsize);
	}
	*/
	for(i=0; i<NFARMERS; i++){
		prc_farmer[i] = create(prod, 1024, 200, "farmer", 5, prod_sema, cons_sema_t, buf,
farmer_tags[i], farmer_grow_times[i]);
```

```
		//kprintf("process start prod\n");

	}

	for( j=0; j<NVEGETARIANS; j++){

		prc_vege[j] = create(cons, 1024, 20 , "vegetarian", 8, j,cons_sema_t, cons_sema,
prod_sema, buf, vegetarian_tags[j], vegetarian_hungers[j], vegetarian_sleep_times[j]);

	}

	//prc_vege[0] = create(cons,  1024, 50 , "vegetarian1",8 , 0, cons_sema_t,
cons_sema,prod_sema, buf, vegetarian_tags[0], vegetarian_hungers[0], vegetarian_sleep_times[0]);

	//prc_vege[1] = create(cons, 1024, 100 , "vegetarian2", 8, 1, cons_sema_t,cons_sema,
prod_sema, buf, vegetarian_tags[1], vegetarian_hungers[1], vegetarian_sleep_times[1]);

	//prc_vege[2] = create(cons, 1024, 150 , "vegetarian3", 8, 2, cons_sema_t,cons_sema,
prod_sema, buf, vegetarian_tags[2], vegetarian_hungers[2], vegetarian_sleep_times[2]);

		//kprintf("process start cons\n");

//	}

	for(i1=0; i1<NFARMERS; i1++){

		resume(prc_farmer[i1]);

		//	kprintf("resume here farmers\n");

	}

	for( i2=0; i2<NVEGETARIANS; i2++){

		resume(prc_vege[i2]);

		//	kprintf("resume here vegetarians\n");

	}

}


/*------------------------------------------------------------------------

 *  stop_farm  -  Stops the currently executing farm simulation

 *------------------------------------------------------------------------
```

```
 */
void stop_farm(void)
{
	//kprintf("start to stop\n");
	semdelete(prod_sema);
	semdelete(cons_sema_t);
	semdelete(cons_sema);
	int i,i1;
	for( i=0; i<NFARMERS; i++){
		kill(prc_farmer[i]);
	}
	for( i1=0; i1<NVEGETARIANS; i1++){
		kill(prc_vege[i1]);
	}
}



/*-----------------------------------------------------------------------
 *  print_farm_report  -  Prints the final report for the farm simulation
 *-----------------------------------------------------------------------
 */
void print_farm_report(void)
{
	//kprintf("start to print\n");
	int num_f[NFARMERS];
```

```
int i,i1,i2,j1,j2,i3,j3,j4;
for(i = 0; i<NFARMERS; i++){
        num_f[i] = 0;
}
int num_v[NVEGETARIANS][NFARMERS];

for(i1 = 0; i1<NVEGETARIANS; i1++){
        for(j1 = 0; j1<NFARMERS; j1++){
                num_v[i1][j1] = 0;
        }
}
//memset( (void*)num_v,NVEGETARIANS*NFARMERS,0);

for(i2 = 0; i2<NVEGETARIANS; i2++){
        int ncount;
        ncount = 0;
        //buffer_t *temp_buf = buf_r[i2];
        //kprintf("a size?%d\n",size(temp_buf));
        while(1){
                char temp;
                temp = result[i2][ncount];
                if(temp == '\0') break;
                //kprintf("%d,%c\n",i2,temp);
                for(j2 = 0;j2<NFARMERS; j2++){
                        if(temp == farmer_tags[j2]){
```

```
                                num_v[i2][j2]++;
                                num_f[j2]++;
                        //              kprintf("%d\n",num_v[i2][j2]);
                        //              kprintf("%d\n",num_f[j2]);
                                };
                        }
                //kprintf("a size?%d\n",size(buf_r[0]));
                //kprintf("b size?%d\n",size(buf_r[1]));
                //kprintf("c size?%d\n",size(buf_r[2]));
                ncount++;
                }
        }
        for(i3 = 0; i3<NFARMERS; i3++){
                kprintf("Farmer %c:sold %d carrots\n",farmer_tags[i3], num_f[i3]);

        }
        for(j3 = 0; j3<NVEGETARIANS; j3++){
                kprintf("Vegetarian %c: ",vegetarian_tags[j3]);
                for(j4 = 0; j4<NFARMERS-1;j4++){
                        kprintf("%d from farmer %c, ", num_v[j3][j4],farmer_tags[j4]);
                }
                kprintf("%d from farmer %c.", num_v[j3][NFARMERS-1],farmer_tags[NFARMERS-1]);
                kprintf("\n");
        }
}
```