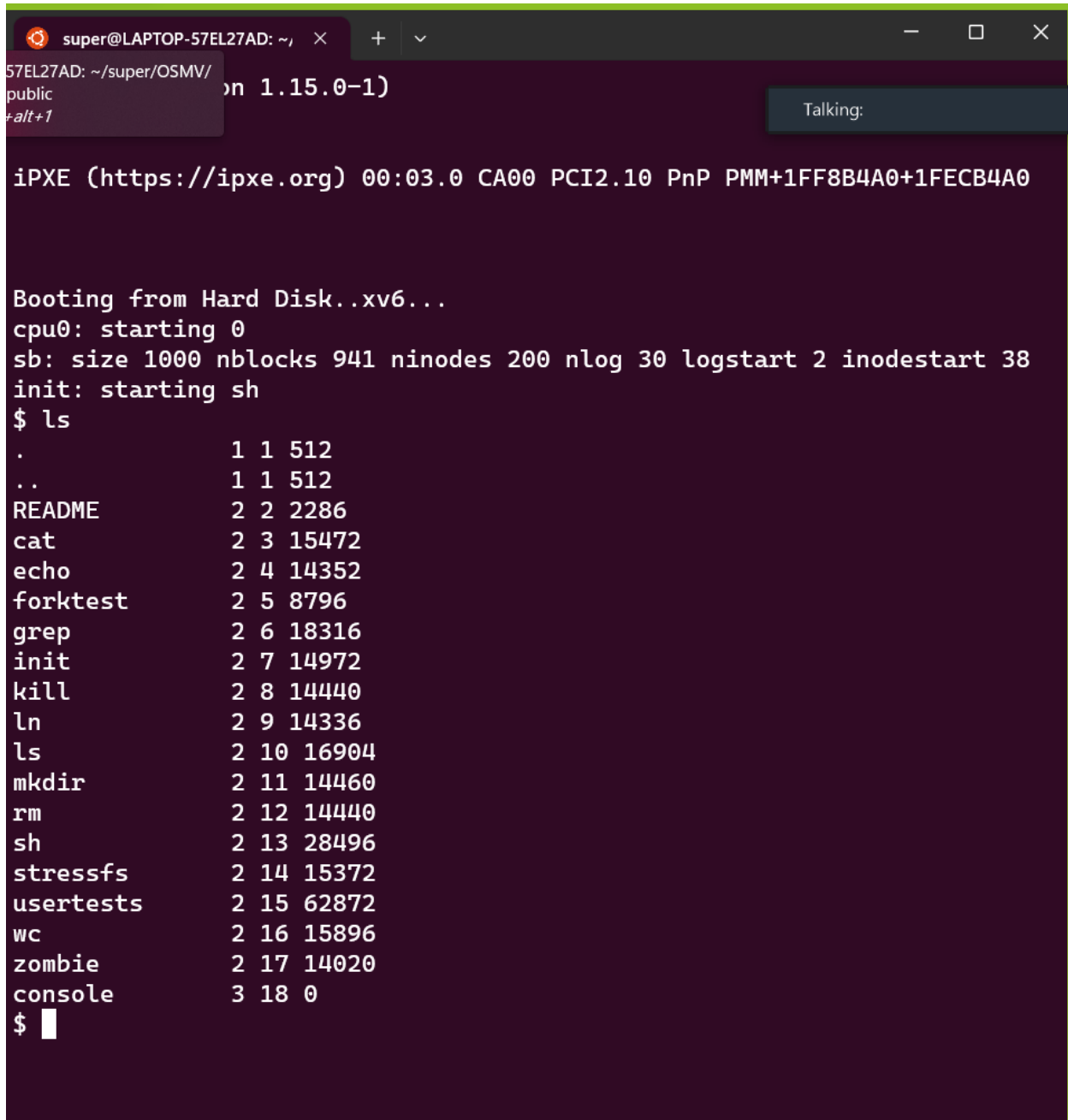
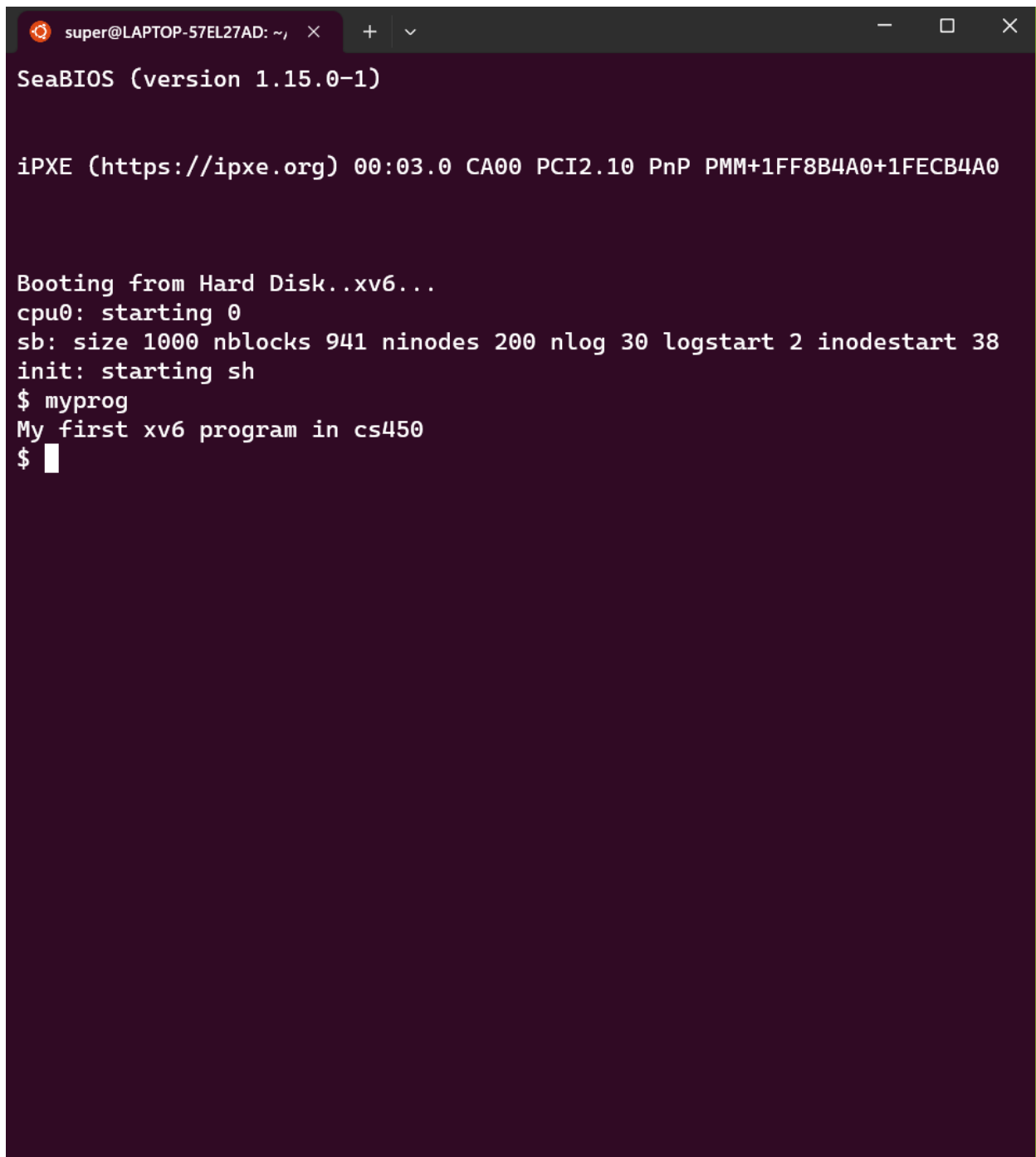


Part A



```
super@LAPTOP-57EL27AD: ~  
57EL27AD: ~/super/OSMV/ on 1.15.0-1)  
public  
+alt+1  
Talking:  
  
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0  
  
Booting from Hard Disk..xv6...  
cpu0: starting 0  
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 38  
init: starting sh  
$ ls  
.  
..  
README  
cat  
echo  
forktest  
grep  
init  
kill  
ln  
ls  
mkdir  
rm  
sh  
stressfs  
usertests  
wc  
zombie  
console  
$
```

A terminal window with a dark purple background and white text. The window title bar shows 'super@LAPTOP-57EL27AD: ~' with standard window controls. The text in the terminal shows the SeaBIOS boot process, including iPXE booting from a hard disk and the start of an xv6 program.

```
super@LAPTOP-57EL27AD: ~  
SeaBIOS (version 1.15.0-1)  
  
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0  
  
Booting from Hard Disk..xv6...  
cpu0: starting 0  
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 38  
init: starting sh  
$ myprog  
My first xv6 program in cs450  
$
```

Part B

Chapter 3 Homework

1. Run process-run.py with the following flags: -l 5:100,5:100. What should the CPU utilization be (e.g., the percent of time the CPU is in use?) Why do you know this? Use the -c and -p flags to see if you were right.

It should be 100% because we are running without stopping.

```
super@LAPTOP-57EL27AD:~/super/OSMV2/ostep-homework/cpu-intro$ ./process-run.py -l 5:100p
Time      PID: 0      PID: 1      CPU      I/Os$ ./process-run.py -l 5:100p
1         RUN:cpu    READY      1
2         RUN:cpu    READY      1
3         RUN:cpu    READY      1
4         RUN:cpu    READY      1
5         RUN:cpu    READY      1
6         DONE      RUN:cpu    1
7         DONE      RUN:cpu    1
8         DONE      RUN:cpu    1
9         DONE      RUN:cpu    1
10        DONE      RUN:cpu    1

Stats: Total Time 10
Stats: CPU Busy 10 (100.00%)
Stats: IO Busy 0 (0.00%)
```

2. Now run with these flags: ./process-run.py -l 4:100,1:0. These flags specify one process with 4 instructions (all to use the CPU), and one that simply issues an I/O and waits for it to be done. How long does it take to complete both processes? Use -c and -p to find out if you were right.

It should take more time than the last one because we are waiting for the I/O.

```
puper@LAPTOP-57EL27AD:~/super/OSMV2/ostep-homework/cpu-intro$ ./process-run.py -l 4:100,1:0 -c
Time      PID: 0      PID: 1      CPU      I/Os
1         RUN:cpu    READY      1
2         RUN:cpu    READY      1
3         RUN:cpu    READY      1
4         RUN:cpu    READY      1
5         DONE      RUN:io     1
6         DONE      BLOCKED    1
7         DONE      BLOCKED    1
8         DONE      BLOCKED    1
9         DONE      BLOCKED    1
10        DONE      BLOCKED    1
11*       DONE      RUN:io_done 1

Stats: Total Time 11
Stats: CPU Busy 6 (54.55%)
Stats: IO Busy 5 (45.45%)
```

3. Switch the order of the processes: -l 1:0,4:100. What happens now? Does switching the order matter? Why? (As always, use -c and -p to see if you were right)

It does matter because we may be waiting less time for the I/O depending on the instructions.

```
puper@LAPTOP-57EL27AD:~/super/OSMV2/ostep-homework/cpu-intro$ ./process-run.py -l 1:0,4:100 -c
Time      PID: 0      PID: 1      CPU      IOs
1         RUN:io    READY      1
2         BLOCKED   RUN:cpu     1
3         BLOCKED   RUN:cpu     1
4         BLOCKED   RUN:cpu     1
5         BLOCKED   RUN:cpu     1
6         BLOCKED   DONE        1
7*        RUN:io_done DONE        1

Stats: Total Time 7
Stats: CPU Busy 6 (85.71%)
Stats: IO Busy 5 (71.43%)
```

4. We'll now explore some of the other flags. One important flag is -S, which determines how the system reacts when a process issues an I/O. With the flag set to SWITCH ON END, the system will NOT switch to another process while one is doing I/O, instead waiting until the process is completely finished. What happens when you run the following two processes (-l 1:0,4:100 -c -S SWITCH ON END), one doing I/O and the other doing CPU work?

Sometimes one is working, and others are blocked, done, busy or ready The CPU waits for the I/O's to finish.

```
S SWITCH ON ENDEL27AD:~/super/OSMV2/ostep-homework/cpu-intro$ ./process-run.py -l 1:0,4:100 -c
Traceback (most recent call last):
  File "/home/super/super/OSMV2/ostep-homework/cpu-intro/./process-run.py", line 299, in <module>
    assert(options.process_switch_behavior == SCHED_SWITCH_ON_IO or options.process_switch_beh)
AssertionError
S SWITCH_ON_END
Time      PID: 0      PID: 1      CPU      IOs
1         RUN:io    READY      1
2         BLOCKED   READY      1
3         BLOCKED   READY      1
4         BLOCKED   READY      1
5         BLOCKED   READY      1
6         BLOCKED   READY      1
7*        RUN:io_done READY      1
8         DONE      RUN:cpu     1
9         DONE      RUN:cpu     1
10        DONE      RUN:cpu     1
11        DONE      RUN:cpu     1
```

5. Now, run the same processes, but with the switching behavior set to switch to another process whenever one is WAITING for I/O (-l 1:0,4:100 -c -S SWITCH ON IO). What happens now? Use -c and -p to confirm that you are right.

It is more efficient as que CPU does not need to wait for the IO's taking a 7 time instead of a 11.

```
S SWITCH_ON_IO -c -p
Time      PID: 0      PID: 1      CPU      IOs
1         RUN:io    READY      1
2         BLOCKED   RUN:cpu     1
3         BLOCKED   RUN:cpu     1
4         BLOCKED   RUN:cpu     1
5         BLOCKED   RUN:cpu     1
6         BLOCKED   DONE        1
7*        RUN:io_done DONE        1

Stats: Total Time 7
Stats: CPU Busy 6 (85.71%)
Stats: IO Busy 5 (71.43%)
```

6. One other important behavior is what to do when an I/O completes. With -I IO RUN LATER, when an I/O completes, the process that issued it is not necessarily run right away; rather, whatever was running at the time keeps running. What happens when you run this combination of processes? (Run ./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH ON IO -I IO RUN LATER -c -p) Are system resources being effectively utilized?

I don't think it is efficient as the general utilization decreased and you can see a cascade of things "READY" waiting for the last process to be "DONE". And some things blocked by the I/O's.

Time	PID: 0	PID: 1	PID: 2	PID: 3	CPU	IOs1
1	RUN:io	READY	READY	READY	1	
2	BLOCKED	RUN:cpu	READY	READY	1	1
3	BLOCKED	RUN:cpu	READY	READY	1	1
4	BLOCKED	RUN:cpu	READY	READY	1	1
5	BLOCKED	RUN:cpu	READY	READY	1	1
6	BLOCKED	RUN:cpu	READY	READY	1	1
7*	READY	DONE	RUN:cpu	READY	1	
8	READY	DONE	RUN:cpu	READY	1	
9	READY	DONE	RUN:cpu	READY	1	
10	READY	DONE	RUN:cpu	READY	1	
11	READY	DONE	RUN:cpu	READY	1	
12	READY	DONE	DONE	RUN:cpu	1	
13	READY	DONE	DONE	RUN:cpu	1	
14	READY	DONE	DONE	RUN:cpu	1	
15	READY	DONE	DONE	RUN:cpu	1	
16	READY	DONE	DONE	RUN:cpu	1	
17	RUN:io_done	DONE	DONE	DONE	1	
18	RUN:io	DONE	DONE	DONE	1	
19	BLOCKED	DONE	DONE	DONE		1
20	BLOCKED	DONE	DONE	DONE		1
21	BLOCKED	DONE	DONE	DONE		1
22	BLOCKED	DONE	DONE	DONE		1
23	BLOCKED	DONE	DONE	DONE		1
24*	RUN:io_done	DONE	DONE	DONE	1	
25	RUN:io	DONE	DONE	DONE	1	
26	BLOCKED	DONE	DONE	DONE		1
27	BLOCKED	DONE	DONE	DONE		1
28	BLOCKED	DONE	DONE	DONE		1
29	BLOCKED	DONE	DONE	DONE		1
30	BLOCKED	DONE	DONE	DONE		1
31*	RUN:io_done	DONE	DONE	DONE	1	
Stats: Total Time 31						
Stats: CPU Busy 21 (67.74%)						
Stats: IO Busy 15 (48.39%)						

7. Now run the same processes, but with -I IO RUN IMMEDIATE set, which immediately runs the process that issued the I/O. How does this behavior differ? Why might running a process that just completed an I/O again be a good idea?

It's a good idea since we are being 100% efficient with the CPU, we do not have to wait anything for the I/O's. Because of this we reduced the time by ten.

```
0,5:100 -S SWITCH_ON_IO -I IO_RUN_IMMEDIATE -c -p
Time    PID: 0      PID: 1      PID: 2      PID: 3      CPU      IOs
1       RUN:io    READY      READY      READY      1
2       BLOCKED  RUN:cpu    READY      READY      1      1
3       BLOCKED  RUN:cpu    READY      READY      1      1
4       BLOCKED  RUN:cpu    READY      READY      1      1
5       BLOCKED  RUN:cpu    READY      READY      1      1
6       BLOCKED  RUN:cpu    READY      READY      1      1
7*      RUN:io_done  DONE      READY      READY      1
8       RUN:io    DONE      READY      READY      1
9       BLOCKED  DONE      RUN:cpu    READY      1      1
10      BLOCKED  DONE      RUN:cpu    READY      1      1
11      BLOCKED  DONE      RUN:cpu    READY      1      1
12      BLOCKED  DONE      RUN:cpu    READY      1      1
13      BLOCKED  DONE      RUN:cpu    READY      1      1
14*     RUN:io_done  DONE      DONE      READY      1
15      RUN:io    DONE      DONE      READY      1
16      BLOCKED  DONE      DONE      RUN:cpu    1      1
17      BLOCKED  DONE      DONE      RUN:cpu    1      1
18      BLOCKED  DONE      DONE      RUN:cpu    1      1
19      BLOCKED  DONE      DONE      RUN:cpu    1      1
20      BLOCKED  DONE      DONE      RUN:cpu    1      1
21*     RUN:io_done  DONE      DONE      DONE      1

Stats: Total Time 21
Stats: CPU Busy 21 (100.00%)
Stats: IO Busy 15 (71.43%)

super@LAPTOP-57EL27AD:~/super/05MV2/ostep-homework/cpu-intro$
```

8. Now run with some randomly generated processes: -s 1 -l 3:50,3:50 or -s 2 -l 3:50,3:50 or -s 3 -l 3:50,3:50. See if you can predict how the trace will turn out. What happens when you use the flag -I IO RUN IMMEDIATE vs. -I IO RUN LATER? What happens when you use -S SWITCH ON IO vs. -S SWITCH ON END?

The fastest configuration will be with the IO on and with the immediate run (because we would use 100% of the CPU). The slowest would be the contrary, the middle point I think would depend on the process and the necessity of the IO.

Chapter 4 Homework

1. Run `./fork.py -s 10` and see which actions are taken. Can you predict what the process tree looks like at each step? Use the `-c` flag to check your answers. Try some different random seeds (`-s`) or add more actions (`-a`) to get the hang of it.

I cannot predict each step. Because the seed randomizes the process. Nevertheless, if I had to guess, I would say it was going to fork rather than exit because of the default `-f` probability but either way I do not know where it would fork.

2. One control the simulator gives you is the fork percentage, controlled by the `-f` flag. The higher it is, the more likely the next action is a fork; the lower it is, the more likely the action is an exit. Run the simulator with a large number of actions (e.g., `-a 100`) and vary the fork percentage from 0.1 to 0.9. What do you think the resulting final process trees will look like as the percentage changes? Check your answer with `-c`.

If we get a `-f` of .1 we will have a tree where almost no nodes have exited the tree. When we do `-f .9` we will get a tree that would be made almost barely by the first action (or couple of actions).

3. Now, switch the output by using the `-t` flag (e.g., run `./fork.py -t`). Given a set of process trees, can you tell which actions were taken?

It is making the fork go in a exclusively cascade effect. We are making each action to be a fork out of the last node created.

4. One interesting thing to note is what happens when a child exits; what happens to its children in the process tree? To study this, let's create a specific example: `./fork.py -A a+b,b+c,c+d,c+e,c-`. This example has process 'a' create 'b', which in turn creates 'c', which then creates 'd' and 'e'. However, then, 'c' exits. What do you think the process tree should look like after the exit? What if you use the `-R` flag? Learn more about what happens to orphaned processes on your own to add more context.

When we erase C and leave D and E orphaned processes, they are going to become child of the first process (A). But when we use the `-R` flag the children of the deleted process will become children of the parent of the deleted. In other words, B is the parent of C, and E, F are the children of C. When we delete C, E & F will become children of B.

5. One last flag to explore is the `-F` flag, which skips intermediate steps and only asks to fill in the final process tree. Run `./fork.py -F` and see if you can write down the final tree by looking at the series of actions generated. Use different random seeds to try this a few times.

Yes, I can follow.

6. Finally, use both `-t` and `-F` together. This shows the final process tree, but then asks you to fill in the actions that took place. By looking at the tree, can you determine the exact actions that took place? In which cases can you tell? In which can't you tell? Try some different random seeds to delve into this question.

I cannot tell just by looking at the trees, I would need the action description because if a process was deleted I wouldn't know if the children were originally part of that process or if they have always been in the place it shows at the "Final Process Tree". I guess that if you turned the probability of exiting down to a minimum then you could know the actions for sure just looking at the tree. Any other way would be mostly guessing.