



*Problem 3:* (34 pts; 20 minutes) Consider the following C code.

```

int main() {
int pid1=0, pid2=0;
int count = 1;

pid1 = fork();

if(!pid1)
    {
    pid2=fork();
    count++;
    }

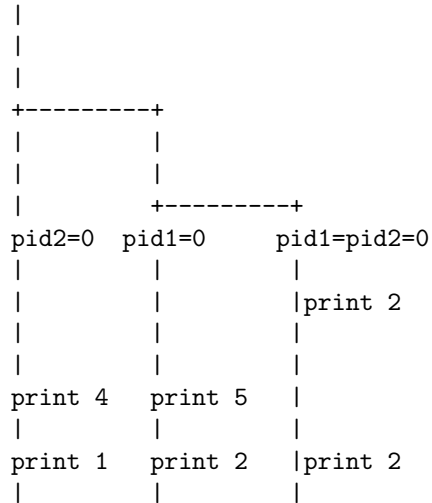
if(pid1==pid2) {
    printf("%d \n", count);
} else
    printf("%d \n", count+3);
}

//For part C, consider what would happen if the following statement is uncommented:
// if(pid2) waitpid(pid2, status, NULL);

printf("%d \n",count);

```

(a) (14 points) Draw the process tree representing the above program execution including outputs. Explain the possible outputs and list two examples (just the numbers that are printed in order; ignore the formatting)



The only restrictions are that each process must print its two numbers sequentially. Otherwise, any interleaving is legal. So, 4-1-5-2-2-2 or 2-4-1-5-2-2, 2-2-4-5-1-2, etc...

(b) (8 points) If the scheduler is non-preemptive and the parent process always runs first after a fork, give one output that is possible and another that is not possible

One 4-1 happens before 5-2 since parent runs first, and 5-2 happens before 2-2 since parent runs first. The schedule is also non-preemptive, which means once a process runs, it continues to the finish or until it has to wait which does not exist other than part c.

Possible: 4-1-5-2-2-2. Not possible: anything else such as 5-2-4-1-2-2 (child1 ran first) or 4-5-1-2-2-2 since the scheduler preempted one process to run the other as we see 4-1 interleave with 5-2.

(c) (6 points) Assume that the commented code marked for part C is uncommented. **Give one output that is possible without the wait that would not be possible with the wait.** In other words, the solution output is possible in the original code, but not possible when the wait is inserted.

The condition here only has the parent process wait for its second child before printing the last number which is 2. However, since the child only prints 2-2 we only get reordering of the 2s, and no visible output that used to be legal will be illegal. More specifically, ignoring 4-1 which can happen in any order, we have 5-2 and 2-2 from the first and second child respectively. The wait forces the 2 in the first child to wait until the 2-2 from the second child prints. So, this means 5-2\*-2-2 and 5-2-2\*-2 with 2\* being the 2 printed by the first child will not occur. However, this sequence can occur 5-2-2-2\* making 5-2-2-2 legal.

Another way to look at it is to see all the possible permutations of 5-2-2-2. They were all legal before provided there was one 2 at the end to preserve the 5-2 sequential order. All those cases remain legal because the 2 could be the 2\* meaning the wait occurred correctly.

So, this ended up being an unintentional trick question. If you explained the effect of the wait you got full credit, otherwise I gave partial credit based on your answer.

(d) (6 points) Consider the original parent process. Explain (pointing to the code where appropriate) where the process could be in the running, ready and waiting states. The scheduler algorithm is unknown and can be anything.

As the parent executes, it is in the running (e.g., printing is in the running state). When it gets preempted and a child runs, then it is in the ready state. If it executes the wait and has to wait for a child, then it is in the waiting state.

*Problem 4:* (28 pts; 15 minutes)

A group of students are studying for the cs153 midterm. At the same time, they are eating the national computer science food (Pizza). Each student grabs a slice of pizza then eats and studies for a while. If there are no slices, they signal the store to deliver more, and then wait until the pizza arrives. The pseudocode is shown below.

```
1.  int slices = 0;
2.  student() {
3.      while (1) {
4.          mutex.wait()
5.          if (slices == 0) {
6.              pizzaStore.signal();
7.              pizzaArrived.wait ();
8.              slices = n; }
9.          slices --;
10.         mutex.signal();
11.         eat();
12.         study(); }
13. }
```

(a) (10 points) Show the pseudocode for a delivery person thread that responds to requests for new pizza such that it can complete the implementation above.

```
delivery() {
pizzaStore.wait();
makeAndDeliverPizza();
pizzaArrived.signal();
}
```

(b) (9 points) What are the initial values for all the semaphores used in the implementation?

**mutex = 1, pizzaStore = 0, pizzaArrived = 0.**

(c) (9 points) Three student threads arrive at the start when there is no pizza. Explain where they will be waiting.

**One student waits at 7 for the pizza to arrive. The other two wait at 4 to get the mutex after the first student releases it at 10.**