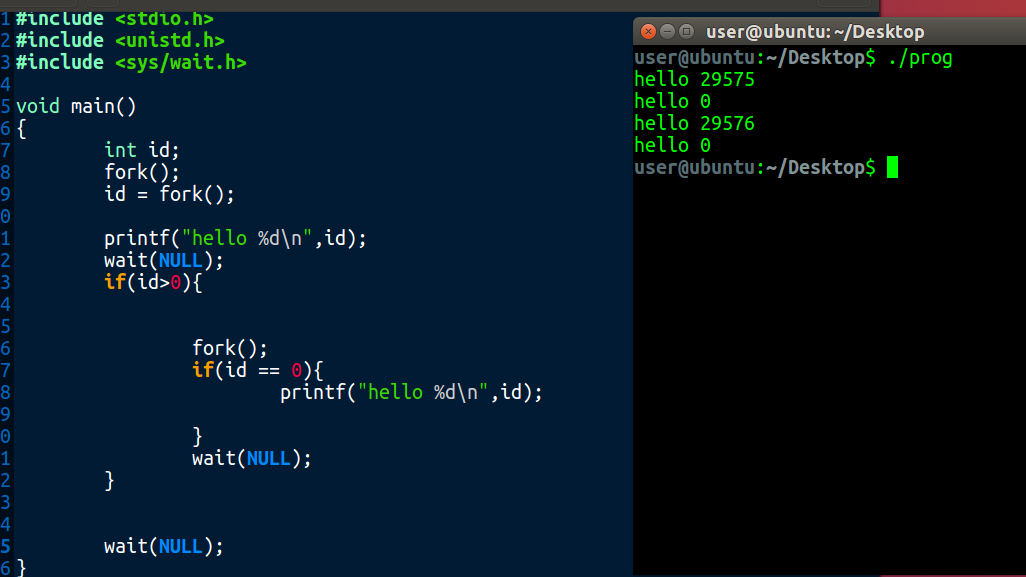
**Assignment 2 Operating Systems**

**Question1.a:**

4 times. At the first print we have 4 processes that have access to the first print, then since we only have 2 parents, only 2 processes will enter the if block, then the second if block will be skipped over because the children that were created in the beginning have not entered the “pid>0” block, therefore in total only 4 hellos will be printed



**Question1.b:**

Fork()

Fork()

Print(hello)

**Question1.c:**

We will use wait() function. The wait() function will cause a parent to wait and let his child process finish running and only when his child is done will the parent continue his own execution, although we should put the wait call in a if block like so: if(id!=0){wait()}

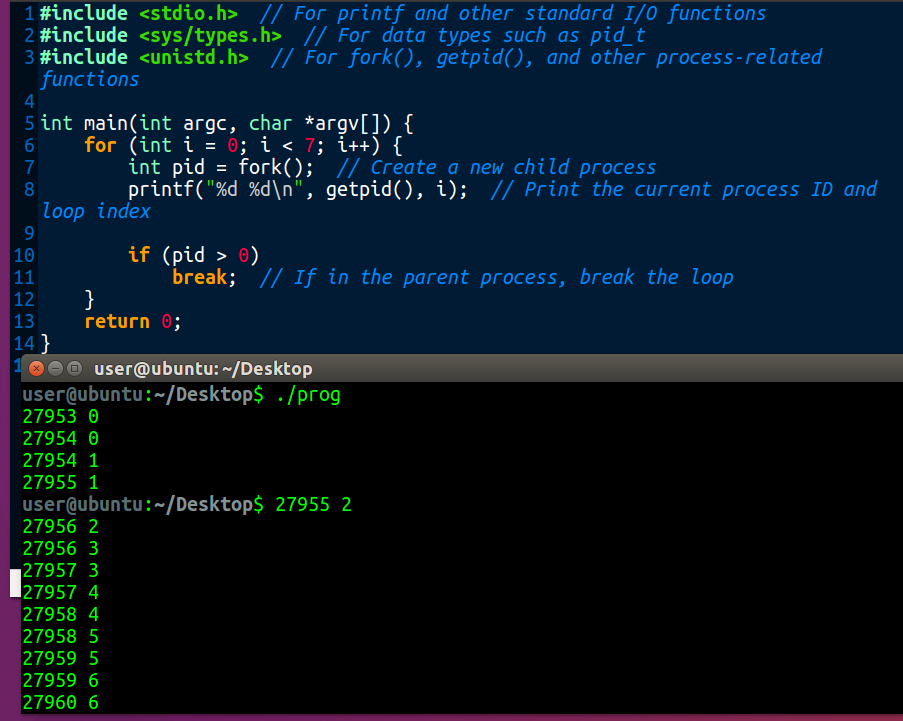
**Question1.d:**

When creating a child process, it creates a copy of its parent’s memory but if it changes any of the variables it will not change the parents’ memory since it’s a copy, same vice versa.

**Question1.d:**

The main difference is the way in which they terminate execution, exit terminates normally while abort terminates abnormally.

**Question2:**

****

**Question2.a:**

**8**

**Question2.a:**

1 : 0

2 : 0

2 : 1

3 : 1

3 : 2

4 : 2

4 : 3

5 : 3

5 : 4

6 : 4

6 : 5

7 : 5

7 : 6

8 : 6

**Question3.a:**

Class cow implements runnable{

Farm farm

Run(){

Farm.eat();

Farm.drink();

Farm.walk();

Farm.milk();

}

}

Class farm{

Semaphore eathouse = semaphore(3)

Semaphore drinkhouse = semaphore(6)

Int fineatdrink = 0

Int milked = 0

Eat(){

eatkhouse.aquire()

eathouse.release()

}

Drink(){

drinkhouse.aquire()

drinkhouse.release()

synchronized(this){

fineatdrink++

notifyall()

}

}

Synchronized walk(){

While(fineatdrink < 4){

Wait()

}

Print(“cow is walking”)

}

Synchronized milk(){

If( !( (thread.currentid %2==0) || milked>=5)){

Return;

}

Print(“cow is milking”)

milked++

}

}

**Question3.b.1:**

wait/signal(notify?): these functions are related to “guarded suspension”. The wait() function causes the thread to enter a waiting room and give back the key if its holding one, the signal() function signals all threads in the waiting room to leave the waiting room and resume their execution.

Tryacquire() function is used to try and acquire a “ticked” from a semaphore, the sem has tickets available then a ticket will be decremented and the thread will be able to execute, if the sem has no tickets the thread will have to wait.

**Question3.b.2:**

the shared object is the tickets. And he needs to be synchronized.

**Question3.b.3:**

yes, and it can be useful in a few scenarios, for example there might be a thread which we would like to have greater weight than other threads.

**Question3.c:**

**See attached java file**

**Question3.d:**

Because we want the semaphores to be shared between all threads, if they were inside each personal thread they would only be relevant to their own thread and not to all threads.

**Question4:**

1. **Create a file descriptor to act as a buffer to communicate between processes(int fd[2],fd[0] is for reading, fd[1] is for writing)**
2. **In parent process call fork(), lets say the child from that fork is child a, within child process use write method to write a message using previous file descriptor(write(fd[1],msg,strlen(msg)))**
3. **Within parent process call fork again, this child will be child b, within child b process use read method to read the message in the buffer using previous file descriptor (read(fd[0],msg,strlen(msg))))**
4. **We also need to close file descriptors in processes which they’re not being used**

**Question5.a:**

**2**

**Question5.b:**

**A parent creates a child, the child exits, and then the parent gets stuck in a loop**

**Question5.c:**

**Child will be in zombie state**