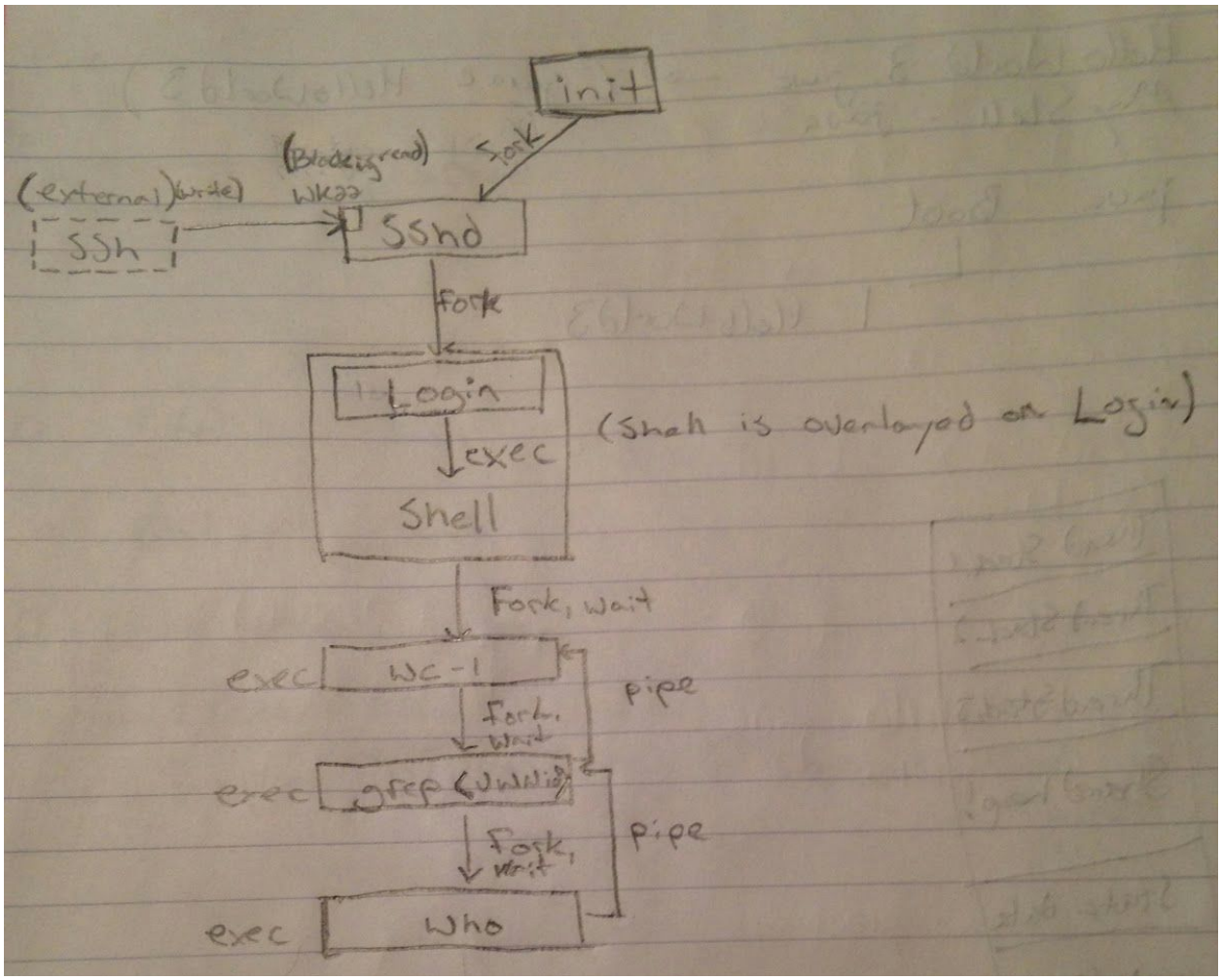
In Unix, the first process is called init. All the others are descendants of “init”. The init process spawns a sshd process that detects a new secure ssh requested connection (WKPort 22). Upon a new connection, sshd spawns a login process that then loads a shell on it when a user successfully logs into the system. Now, assume that the user types

who | grep <uwnetid> | wc –l

Draw a process tree from init to those three commands. Add fork, exec, wait, and pipe system calls between any two processes affecting each other.



3.3 When a process creates a new process using the fork() operation, which of the following states is shared between the parent process and the child process?

a. Stack

b. Heap

c. Shared memory segments

3.9 Including the initial parent process, how many processes are created by the program shown in Figure 3.34?

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    /* fork a child process */
    fork();

    /* fork another child process */
    fork();

    /* and fork another */
    fork();

    return 0;
}
```

Figure 3.34 How many processes are created?

There are 8 processes. The first fork creates a child, and with the parent there are 2. The second fork creates a child for both the first child and the parent, so then there are 4. Finally the third fork creates a child for each of the previous 4 processes, leaving 8 total.

3.10 Using the program in Figure 3.35, identify the values of pid at lines A, B, C, and D. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid, pid1;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) { /* child process */
      pid1 = getpid();
      printf("child: pid = %d",pid); /* A */
      printf("child: pid1 = %d",pid1); /* B */
   }
   else { /* parent process */
      pid1 = getpid();
      printf("parent: pid = %d",pid); /* C */
      printf("parent: pid1 = %d",pid1); /* D */
      wait(NULL);
   }

   return 0;
}
```

Figure 3.35 What are the pid values?

A: 0

B: 2603

C: 2603

D: 2600

3.12 Explain the fundamental differences between RMI and RPCs.

RMI is the Java proximation of the RPC. RMI stands for Remote Method Invocation and RPC stands for Remote Procedure Call. According to DifferenceBetween.com:

"The basic difference between RPC and RMI is that RPC is a mechanism that enables calling of a procedure on a remote computer while RMI is the implementation of RPC in java. RPC is language neutral but only supports primitive data types to be passed. On the other hand, RMI is limited to Java but allows passing objects. RPC follows traditional procedural language constructs while RMI supports object-oriented design." (http://www.differencebetween.com/difference-between-rpc-and-vs-rmi/)