Problem 3.1

a) Question:

Locate where in system/ the source code of nulluser() is defined.

Answer:

The code for nulluser() is found in the initialize.c file.

b) Question:

Find out where in the XINU source code this ancestor of all processes is located and what name, i.e., process ID (PID), it assigns itself.

Answer:

The code that initializes the ancestor of all process is located in initialize.C as well, inside the sysinit() function. It assigns itself PID = 0 (from process.h) and its name is prnull.

c) Question:

What does the ancestor process running the function nulluser() do for the rest of its existence?

Answer:

The code itself becomes the ancestor process after initialization. It remains ready to execute, so it runs indefinitely without being killed.

d) Question:

Does nulluser() ever return after being called by the code in start.S?

Answer:

No, it enters an infinite loop that prevents it from returning after being called by the code in start.S.

e) Question:

halt() is called after nulluser() in start.S. Where is the source code of halt() located and what does it do?

Answer:

The halt() source code is located in intr.S file. It halts the system forever. It enters a infinite loop that doesn't do anything.

f) Question:

What happens if you remove this function call from start.S? Does XINU run as before?

Answer:

Yes, XINU runs as before.

g) Question:

What happens if you replace the while-loop at the end of nulluser() with a call to halt()? Discuss all your findings.

Answer: Based on my observations, XINU works exactly the same. However, during class, Professor told us that when kernel halts() the system it uses all of its cpu. So if null process halts the system, shell shouldn't be able to run. My guess is that hall doesn't halts the system it just makes a part of it busy doing nothing.

Problem 3.2

a) Question:

What happens in Linux after a new process is created using fork()? Note, create() specifies what code to run as the child process through its first argument (a function pointer).

Answer:

Fork() takes no arguments and it return a process ID. If the process ID < 0, it means that the child was not created. If process ID == 0, it means that that process is the child. And if the process ID > 0, it means that the process is the parent and the return ID is the child's process ID.

When the child is successfully created, Unix will make two copies of the address space. One for the parent process and one for the child. Then, they both will start execution at the next instruction following the fork() call.

b) Question:

Determine who runs first: parent or child. Try to empirically gauge the answer by running test code on the frontend Linux PCs. (We will discuss scheduling of processes in modern operating systems, a complex topic, when discussing process management.)

Answer:

Sometimes parent runs first and sometimes child does it. It seems there is no arbitrary order. There could be also a priority based runtime such as in XINU.

c) Question:

As an app programmer, do you have a preference as to which process -- parent or child -- should run next in Linux? Explain your reasoning.

Answer:

In general, I don't have any preference to which process runs first. However, depending on the program I am writing, there are some cases in which I do prefer or need one process to run first. It all depends on the program I am writing.

e) Question:

How is newProcess() fundamentally different from the way XINU's create() works?

Answer:

They both require a file path as an argument and create a new process that runs that file. However, one difference is that the process that create() creates is in a suspended state, without using any computer cycles untill it runs resume(). On the other hand, the process created by fork() is already using computer cycles from the moment it is created.

f) Question:

How does Linux's clone() compare to XINU's create()?

Answer:

The child process that Linux's clone() creates, can share some of its execution context with the calling process

g) Question:

How does Linux's posix_spawn() compared to your newProcess() implementation?

Answer:

They are really similar. They both create a child process that it's next execution is the executable file given as an argument. However posix_spawn() was built for machines that are not capable of handleing fork(2), therefore its implimentation is without using fork(), while newProcess()'s implementation does use fork().

h) Question:

Is there a best way to create processes? Explain your reasoning.

Answer:

Not really. It depends on the situation and on the program we want to code. For instance if you are using a machine that it is not powerfull enough to run fork(2)

your best option is to use posix_spawn. So the best option is always dependent on the situation.

3.4)

I did the extra credit part.