In order for multiple processes to be able to run on the same computer, using a shared address space, we need to have an operating system that controls which process gets to run at any given point in time and ensures that the state that is currently loaded into the system is the state of the current process.

Through the following example we will take a closer look at how the operating system for the beta works.

We begin by examining the code responsible for maintaining the current processor state as well as scheduling which process should run at any given point in time.

The operating system uses a structure called MState to keep track of the value of the 32 registers for each of the running processes.

The UserMState variable holds the state of the currently running process.

The ProcTbl or process table, holds all the necessary state for each of the processes that runs on the machine.

For each process, it stores the value of all of its registers in the State variable.

Additional state may be stored per process.

Examples of some additional state that would go in a process table are a page map per process if we want to use virtual memory.

Another example, is a keyboard identifier, that associates hardware with a particular process.

The Cur, or current variable, holds the index of the currently running process.

When we want to switch control from one process to another, we call the Scheduler.

The Scheduler first stores the state of the currently running process into the process table.

Next, it increments the current process by 1.

If the current process was process N – 1, then it goes back to process 0.

Finally, it reloads the user variable with the state for the new current process.

In order to be able to run a diagnostic program on one process that samples the values in the PC of another process, a supervisor call named SamplePC is provided for you.

The C portion of the SVC handler is provided for you here.

It is incomplete, so our first goal is to determine what should replace the ???.

We are told that the way SamplePC SVC works is that it takes a process number p in R0, and returns in R1 the value currently in the program counter of process p.

The handler shown here reads register 0 from the UserMState data structure and stores the value into variable p.

This is the number of the process that is to be monitored.

In order to determine the value of the PC of process p, one can look up the value of the XP register that was saved for process p the last time process p was run.

The XP register holds the value of the next PC address.

So reading the XP register from ProcTbl[p] tells us the next value of the pc for process p.

The pc value is to be returned in register R1 of the current program.

This means that the missing code is UserMState.Regs[1] and that should be assigned the value of pc.

Suppose you have a compute-bound process consisting of a single 10,000 instruction loop.

You use the SamplePC supervisor call to sample the PC while running this loop.

You notice many repeated values in the results of the SamplePC code.

You realize that the reason this is happening is because every time your profiling process gets scheduled to run, it makes many SamplePC calls but the other processes aren't running so you are getting the same sampled PC multiple times.

How can the repeated samples be avoided?

To avoid repeated samples, we add a call to Scheduler() in our SamplePC handler.

This ensures that every time that the profiler process is scheduled, it only samples a single PC value and then let's another process run.

Suppose that you continue using the original version of your SamplePC handler, which does not call Scheduler, and you use it to test this code which repeatedly calls the GetKey() supervisor call to read a character from the keyboard, and then calls the WrCh() supervisor call to write the character that was just read.

We want to answer the question of which PC value will be the one reported the most often.

Address 0x100 which is the address of the GetKey() call is reported the most often because most of the time when GetKey() is called, there is no key stroke waiting to be processed.

This means that the GetKey() call will get processed over and over again until there is finally a key to process.

The result of this is that the PC value that shows up the most often is 0x100.

The last question we want to consider is what behavior is observed when the profiler which is running in process 0 profiles process 0 itself.

Assume that the profiler code consists primarily of one big loop which has a single call to the SamplePC supervisor call at instruction 0x1000.

The rest of the loop processes the data that was collected by the SamplePC call.

The question we want to answer is what is observed in the SamplePC results.

We are given 4 choices to select from.

The first choice to consider is whether or not all the sampled PC values point to the kernel OS code.

This choice is false because if it were true it would mean that you somehow managed to interrupt kernel code which is not allowed by the beta.

The next choice to consider is whether the sampled PC is always 0x1004.

This seems like it might be a correct choice because the SamplePC supervisor call is at address 0x1000, so storing PC + 4 would result in 0x1004 being stored into the UserMState.Regs[XP].

However, if you look closely at the SamplePC handler you see that the XP register is read from the ProcTbl.

But the UserMState regs are only written to ProcTbl when Scheduler() is called, so reading the value from ProcTbl would provide the last value of the PC when process 0 was last interrupted.

To get the correct value, you would need to read UserMState.Regs[XP] instead.

The third choice is that the SamplePC call never returns.

There is no reason for this to be true.

Finally, the last choice is "None of the above".

Since none of the other choices were correct, this is the correct answer.