WE17.1

Bookmark this page

Video explanation of solution is provided below the problem.

Operating Systems

4 points possible (ungraded)

BetaSoft, Inc, the leading provider of Beta OS software, sells an operating system for the Beta similar to that described in lecture. It uses a simple round-robin scheduler, and has no virtual memory -- all processes share a single address space with the kernel. The OS timeshares the Beta CPU among N processes using a simple, familiar scheduler shown below.

```
struct MState { int Regs[31]; } UserMState;

struct PCB {              // Process Descriptor Block
struct MState State;      // Saved process state
...;                      // Possible other stuff
} ProcTbl[N];             // One PCB per process

int Cur = 0;

Scheduler() {
  ProcTbl[Cur].State = UserMState;
  Cur = (Cur+1) % N;
  UserMState = ProcTbl[Cur].State;
}
```

Several of Betasoft's customers use the Beta for long, compute-bound applications, and have asked for a tool to help them find where their programs are spending most of their time. To accommodate these requests, BetaSoft has implemented a supervisor call, SamplePC, which allows a diagnostic program running in one process to sample the values in the PC of another. Betasoft proposes to write such a program, called a profiler, that takes many samples of PC values from a running program and produces a revealing histogram.

The SamplePC SVC takes a process number p in R0, and returns in R1 the value currently in the program counter of process p. The C portion of the SVC handler is given below:

```
SamplePC_h() {
 int p = UserMState.Regs[0];
 int pc = ProcTbl[p].State.Regs[XP];
 ??? = pc;          // incomplete code!
}
```

1. Give the missing code fragment shown above as ???. Do not include any white space in your response.

   Answer: UserMState.Regs[1]

   Explanation
   UserMState.Regs[] is the kernel data structure that holds the user registers during interrupts or supervisor calls. When Scheduler() is called, it stores the UserMState.Regs[] data structure into the ProcTbl data structure so that it can then load a different processes's registers into the UserMState data structure. The SamplePC handler reads the process number that was passed in register R0 by accessing UserMState.Regs[0]. It then determines the value that is currently in the program counter of process p by looking at the ProcTbl data for process p. Recall that the program counter of an interrupted process is stored in the XP register. So looking at the value of the XP register will provide the desired program counter value. This result then needs to be returned in register R1, so setting UserMState.Regs[1] to pc will return the correct value in the user's register R1 because their UserMState data structure will be copied back into the user's registers when returning to the interrupted program.

   BetaSoft writes a simple profiler using the above SVC and uses it to measure a compute-bound process consisting of a single 10000-instruction loop. Noticing a surprisingly large number of repeated values in the sampled PC data, they cleverly deduce that their profiler is making many SamplePC calls during each time quanta for which the profiling process is scheduled, returning redundant samples from the process being measured.

2. Does adding a call to Scheduler() to the **SamplePC_h** code eliminate the observed problem?

- ◯ Yes ✔
- ◯ No

BetaSoft ignores your solution (keeping the original **SamplePC_h** code), arguing that they'll just collect enough samples that the redundant values won't affect the histogram significantly. They produce a working profiler program that takes many samples of another process's PC and produces a histogram showing code "hot spots". Although the profiler proves useful on compute-intensive application code, BetaSoft tries running it on a simple echo loop running in a process:

```
                    | echo loop, as a test for profiler tool:
      .=0x100       | Test program starts at hex 100
loop: GetKey()      | SVC: read char into R0
      WrCh()        | SVC: type char from R0
      BR(loop)      | ... and keep doing it!
```

3. When run on the above process, what does the profiler report as the most common value of the PC? Answer "NONE" if you can't tell.

Often-reported PC value, or "NONE": 0x

100 Answer: 100

Explanation
This code calls the GetKey() supervisor call in order to try and receive a character from the keyboard. If there is no character available, then the GetKey() supervisor call will get called over and over again until a key is available. This means that most of the time spent in this process is at the GetKey() call which is at address 0x100.

The final BetaSoft profiler program itself is mostly a big loop, consisting of a single SamplePC SVC instruction located at **0x1000**, plus lots of compute-intensive additional code to appropriately format the collected data and write it into a file. Out of curiosity, BetaSoft engineers run the profiler in process 0, and ask it to generate a histogram of sampled PC values for process 0 itself.

4. Which of the following best summarizes their findings?

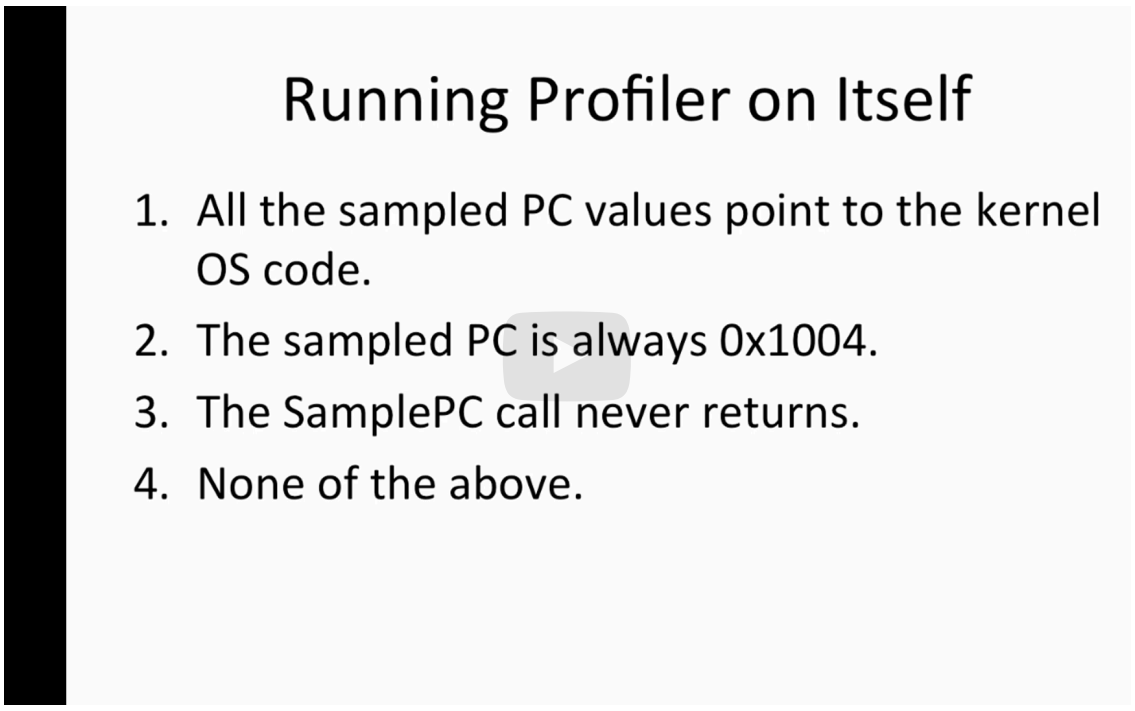
- ◯ All of the sampled PC values point to kernel OS code.
- ◯ The sampled PC is always **0x1004**.
- ◯ The SamplePC call never returns.
- ◉ None of the above. ✔

Explanation
Choice 1: "All of the sampled PC values point to kernel OS code." - is not correct because this would mean that somehow you managed to interrupt kernel code which is not allowed.
Choice 2: "The sampled PC is always 0x1004." - seems like it might be correct since the SamplePC SVC instruction 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 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.
Choice 3: "The SamplePC call never returns." - there is no reason for this to be true.
Choice 4: "None of the above." - Since this is the only remaining choice, this is the correct answer.

Submit

Answers are displayed within the problem

Operating Systems



7:15 / 7:15 1.0x

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]

Video
Download video file

Transcripts
Download SubRip (.srt) file
Download Text (.txt) file

Discussion

Topic: 17. Virtualizing the Processor / WE17.1

Hide Discussion

Add a Post

Show all posts by recent activity

? Can't understand how does SamplePC_h routine work... 4

Previous Next

edX

About
Affiliates
edX for Business
Open edX
Careers
News

Legal

Connect