Graduate Systems
Practice Problems (midsem)

Q.1. Imagine a program with three threads—T1, T2, and T3—that run functions Func1, Func2, and Func3 respectively. There are two locks, LockA and LockB. Initially, both locks are free. Now, thread T2 acquires LockA and then calls `fork` (resulting in a parent process with PID 100 and a child process with PID 200).

(a) In the child process, what will happen to the threads, and what will be the state of the locks?
Ans:

- For the thread:
  - T2: The child process will contain only the thread called `fork` (T2). This thread's state (its registers, call stack, and program counter) is duplicated as it was at the moment of the fork. Essentially, T2 in the child continues execution from just after the fork call, with its state preserved.
  - T1 and T3: These threads are not carried over into the child process. They simply do not exist in the new process.
- For the locks:
  - LockA: Remains locked in the child process because it was held by T2 at the time of the fork.
  - LockB: Remains unlocked as it was free at the time of the fork.

Q.2. Imagine the same program setup (refer Q.1.) with threads T1, T2, and T3 running functions Func1, Func2, and Func3, and two locks, LockA and LockB. This time, however, before thread T2 calls fork, thread T3 has already acquired LockB. In the child process created by T2's fork, which thread(s) will exist, and what will be the state of the locks?

Ans.

Threads in the Child Process:

(i) Only T2 will exist.
The child process is created as a copy of the parent's state at the moment of the fork, but only the calling thread (T2) is duplicated. Threads T1 and T3 will not be present in the child.

Lock States in the Child Process:

(ii) LockA: If T2 held LockA at the time of the fork, it remains locked in the child process.

(iii) LockB: Even though T3 holds LockB in the parent, the child process inherits a copy of the parent's memory, including the lock state. Therefore, LockB will still

appear as locked in the child process—even though the thread (T3) that held it is not present. This can potentially lead to issues like deadlock since no thread in the child process can release LockB.

Q.3. Prepare for conceptual questions (similar to Q.1. and Q.2.) with respect to file handling in child processes and threads

Q.4. Consider the following CPU instructions found in modern CPU architectures like x86. For each instruction, state if you expect the instruction to be privileged or unprivileged, and justify your answer. For example, if your answer is “privileged”, give a one-sentence example of what would go wrong if this instruction were to be executable in an unprivileged mode. If your answer is “unprivileged”, give a one-sentence explanation of why it is safe/necessary to execute the instruction in unprivileged mode.

(a) Instruction to write into the interrupt descriptor table register.

Ans: Privileged, because a user process may misuse this ability to redirect interrupts of other processes.

(b) Instruction to write into a general-purpose CPU register.

Ans: Unprivileged, because this is a harmless operation that is done often by executing processes.

Q.5. Consider a guest VM running on the QEMU/KVM hypervisor. Assume that the guest has just exited.

(a) How does the CPU obtain the address of the KVM code in the host to jump to upon an exit?
Ans: VMCS
(b) After control returns to KVM, from which memory location/data structure/CPU register does KVM obtain the reason for the VM’s exit?
Ans: VMCS
(c) Does (and, can) the QEMU userspace process see all exits of a VM run by it? Answer yes/no and justify. If you answer yes, explain why QEMU needs to see all VM exits. If you answer no, give an example of a VM exit that is not visible in QEMU userspace.
Ans: No, for example, Timer interrupt / Keyboard interrupt / Page fault
(d) If a VM exit needs to be handled by QEMU, from which memory location/data structure/CPU register does QEMU obtain the reason for the VM’s exit?
Ans: Struct KVM RUN

Q.6. Suppose you are trying to build a trap-and-emulate VMM to virtualize an OS on the x86 architecture, using a technique like binary translation. The VMM runs the guest OS at a lower privilege level, while it runs itself at the highest privilege level of ring 0. Now, while the CPU instruction that writes to the “idtr” register (interrupt descriptor table base register) is a privileged instruction, the instruction that reads the idtr is an unprivileged instruction that executes correctly in any privilege level without trapping to ring 0. Should the VMM replace the instruction

to read the idtr by something that traps to the VMM (and can be emulated by the VMM) during binary translation? Answer yes/no and justify. If you answer yes, explain what would go wrong if this instruction is executed directly by a virtualized guest without trapping to the VMM. If you answer no, explain why this instruction is safe to execute by a virtualized guest.

Ans: Yes, VMM should replace the instruction and not let the guest directly read the IDTR. This is because the guest OS would have replaced the IDTR, and the guest OS will panic after finding out that the value it set is not the same as the value it read from the IDTR.

Q.7. Consider an architecture that uses 3-level hierarchical paging. Suppose the hypervisor runs 10 guest VMs with 5 active processes each.

→ Variant of this question is part of midsem

How many extra page tables does the hypervisor maintain to virtualize the guest, in addition to the page tables maintained by the guest VM itself?

Answer this question for (a) Extended Page Table (EPT) are used; (b) Shadow page tables are used.

Ans. (a) 10 EPT (one EPT per guest VM; EPT maps GPA to HPA)

(b) 5 * 10 = 50

(One shadow page table per active process maps GVA to HPA, and there will be 10*5 active processes in the Guest VMs)