1. In Linux, as in most operating systems, all the general-purpose registers (including RIP) are saved on the process’s kernel stack (see slides XIX-14 and XIX-35) when a trap (such as a system call) or interrupt takes place, then restored on return from the trap or interrupt, at which point the process continues execution as if nothing had happened. However, if, while handling the trap or interrupt, the OS notices that the process has a pending unmasked signal, it must arrange so that on return to the process, rather than resuming execution where it was when the trap or interrupt took place, the process instead enters the signal handler. When the signal handler returns, the process (finally) resumes execution where it was when the trap or interrupt took place.

The OS makes all this happen by, just prior to returning to the process after the trap or interrupt, pushing some information onto the process’s user stack and, rather than returning to the instruction at the saved value of RIP, sets RIP to the first instruction of the signal handler. This causes the process to resume execution at the beginning of the signal handler.

What information does the OS push onto the process’s user stack to make all this happen? (In particular, once the signal handler returns, the process’s execution must (soon) resume where it was when the trap or interrupt took place.) The user process may have to invoke additional code to perform certain operations. You can assume the address of this code is known. What does this code need to do? (You don't have to supply the code). The signal handler itself is an ordinary C function — it doesn’t do anything special. Hint: the code that the signal handler returns to isn’t necessarily the code that was being run at the time of the trap or interrupt.