OS Pset4 #1 and #2

Thodori Kapouranis, Brian Doan and Philip Blumin

1A) What key sequence would you use to cause a foreground process to terminate and dump core (if possible)?

If we look at the table of signal numbers and their default dispositions (Unit 4, pg2), we can see that signal 'SIGQUIT' ("Keyboard Interrupt") has a default disposition of terminate+coredump. To send this signal to a foreground process, we would do the key combination ' ctrl+\ '.

1B) When would a signal SIGTTOU be sent to a process? What happens to the process when it receives it (assuming that this signal's disposition is default)?

SIGTTOU would be sent if a background process tries to write to the terminal or sets its modes. This only happens if the terminal is in 'tostop' mode, which is set by the cmd 'stty tostop'. The default disposition of SIGTTOU is set to be 'stop'. Stop is used for job control, and can be resumed by the signal SIGCONT.

1C) Process "A" sends (using the kill system call) signal #40 to process "B". It does so 3 times in a row. At the time, process "B" has signal #40 in the blocked signals mask and has set up a handler for this signal. At some later time, process "B" unblocks signal #40. What happens and why?

Since the signal number is above 32, we know that this is a **real time signal**. What makes real-time signals different from the rest is that their delivery does not simply toggle a pending bit like traditional signals, but they get added onto a queue. This means that unlike a single bit, multiple instances of the delivered signal can be recorded in the order they came in.
When the process unblocks signal #40, the first signal to arrive causes the handler call. The handler will block that signal's bit, do its handler function, and if it returns, unblocks the bit. This allows the second to arrive signal to be handled, and if it also is successful, the third signal will then get handled.

1D) What does it do? What is its wait exit status? Don't just tell me, explain what is happening and why

1. First, we initialize a jmp_buf name wormhole.
2. In main, we creates a null pointer *p
3. The signal system call changes the disposition of 'SIGSEGV' to be sent to our handler 'handler()'
4. After this, we set the jump point with the if statement. Since calling setjmp() by itself returns 0, this if block is not executed.
5. *p-- is an illegal action on a null ptr, and causes SIGSEGV to generate.
6. This causes the handler previously assigned to SIGSEGV to be called
    a. First, when the handler is called it blocks signal bit #11 to avoid reentrant calls
    b. i is initialized to 0, iterated so now i=1
    c. prints "In handler Instance 1" to stdout
    d. longjmp(wormhole,i) means go to where we setjmp(wormhole), and make it return i
7. Since i=1, the contents inside the if(setjmp(wormhole)) block get executed.
    a. *p-- is an illegal action on a null ptr, and causes SIGSEGV to generate
    b. Since the handler was called but the handler itself never exited (it longjmp'd), it never unblocked the signal bit for SIGSEGV.
    c. If a SIGSEGV is generated while its bit is blocked, this results in undefined behavior('NOTES' in man 2 sigprocmask). The kernel then decides to terminate the program with SIGSEGV.
    d. The returning exit code will be 139 because the program was exited by SIGSEGV(#11).

2A. Can this program ever produce 'ABA'

No: Each write call to the writing end of the pipe is done **atomically** in up to 4k chunks. Since each write system call here is 1K blocks of the char argument passed to the child() function, each child atomically writes 1K. This means a 'B' can **never** somehow make it in between two A's, because an atomic process is guaranteed to finish running with no other process happening in between it finishes.

2B. What happens if you remove the line marked 'THIS LINE'

This line refers to the parent closing its writing end of the pipe. This means that the parent will have a **dangling file descriptor to the write end of the pipe**, and when the children exit, this will be the only file descriptor pointing to the write end. So, an EOF is never sent through the pipe to the reader (the parent itself), and it is stuck in an indefinite interruptible sleeping state (infinite wait).