Step through the crash

Now that we understand why the `do-nothing` program crashes, we will use GDB to step through the execution of the `do-nothing` test in Pintos, starting from when the kernel boots. Our goal is to find out how we can modify the Pintos user program loader so that `do-nothing` does not crash, while becoming acquainted with how Pintos supports user programs. To do this, change your working directory to `proj-pregame/src/userprog/` and run

```
FORCE_SIMULATOR=--bochs PINTOS_DEBUG=1 pintos-test do-nothing
```

Side note: You can alias this by adding

```
alias pintos-debug='FORCE_SIMULATOR=--bochs PINTOS_DEBUG=1 pintos-test'
```

to `~/.bashrc` and you will be able to run GDB with `pintos-debug do-nothing`.

GDB should now be open. At a high level, the following must happen before Pintos can start the `do-nothing` process:

- The BIOS reads the Pintos bootloader (`proj-pregame/src/threads/loader.S`) from the first sector of the disk into memory at address `0x7c00`.
- The bootloader reads the kernel code from disk into memory at address `0x20000` and then jumps to the kernel entrypoint (`proj-pregame/src/threads/start.s`).
- The code at the kernel entrypoint switches to 32-bit protected mode 1 and then calls main (`proj-pregame/src/threads/init.c`).
- The main function boots Pintos by initalizing the scheduler, memory subsystem, interrupt vector, hardware devices, and file system.

You're welcome to read the code to learn more about this setup, but you don't need to understand how this works for the Pintos projects or for this class.

Set a breakpoint at `run_task` and continue in GDB to skip the setup. As you can see in the code for `run_task`, Pintos executes the `do-nothing` program (specified on the Pintos command line), by

invoking

```
process_wait(process_execute("do-nothing"));
```

from `run_task`. Both `process_wait` and `process_execute` are in `proj-pregame/src/userprog/process.c`.

Now, answer the following questions:

1. Step into the `process_execute` function. What is the name and address of the thread running this function? What other threads are present in Pintos at this time? Copy their `struct thread`s. (Hint: for the last part, `dumplist &all_list thread allelem` may be useful.)

2. What is the backtrace for the current thread? Copy the backtrace from GDB as your answer and also copy down the line of C code corresponding to each function call.

3. Set a breakpoint at `start_process` and continue to that point. What is the name and address of the thread running this function? What other threads are present in Pintos at this time? Copy their `struct thread`s.

4. Where is the thread running `start_process` created? Copy down this line of code.

5. Step through the `start_process` function until you have stepped over the call to load. Note that load sets the `eip` and `esp` fields in the `if_` structure. Print out the value of the `if_` structure, displaying the values in hex (hint: `print/x if_`).

6. The first instruction in the `asm volatile` statement sets the stack pointer to the bottom of the `if_` structure. The second one jumps to `intr_exit`. The comments in the code explain what's happening here. Step into the `asm volatile` statement, and then step through the instructions. As you step through the `iret` instruction, observe that the function "returns" into userspace. Why does the processor switch modes when executing this function? Feel free to explain this in terms of the values in memory and/or registers at the time `iret` is executed, and the functionality of the `iret` instruction.

7. Once you've executed `iret`, type `info registers` to print out the contents of registers. Include the output of this command on Gradescope. How do these values compare to those when you printed out `if_`?

8. Notice that if you try to get your current location with `backtrace` you'll only get a hex address. This is because because the debugger only loads in the symbols from the kernel. Now that we are in userspace, we have to load in the symbols from the Pintos executable we are running, namely `do-nothing`. To do this, use `loadusersymbols tests/userprog/do-nothing`. Now, using `backtrace`,

you'll see that you're currently in the `_start` function. Using the `disassemble` and `stepi` commands, step through userspace instruction by instruction until the page fault occurs. At this point, the processor has immediately entered kernel mode to handle the page fault, so `backtrace` will show the current stack in kernel mode, not the user stack at the time of the page fault. However, you can use `btpagefault` to find the user stack at the time of the page fault. Copy down the output of `btpagefault`.
