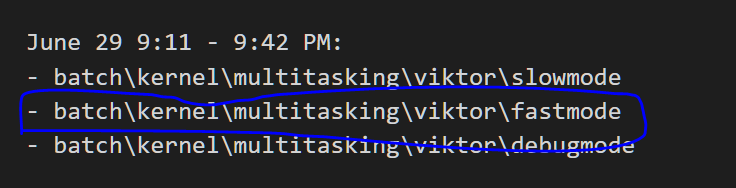
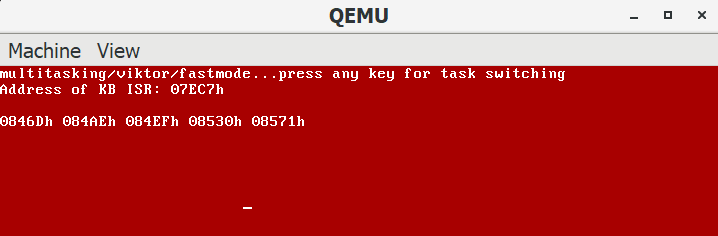
LDE – Multitasking – OSTEP

September 23, 2021

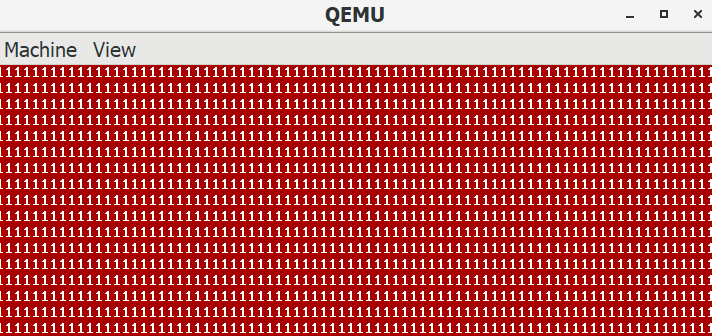
desktop\osdev\sources\notes.txt



Open dos to that directory and go ahead and run the program. You will see the following screen if you run in on qemu or bochs



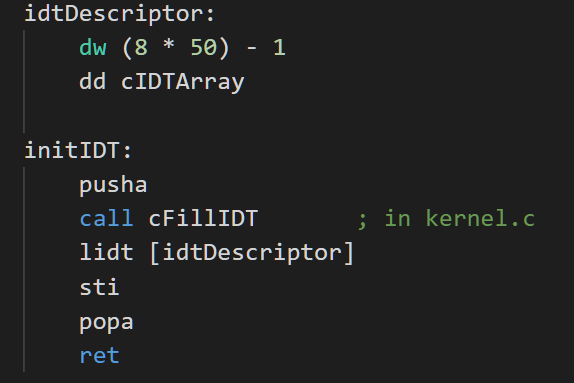
As you *press and release* keyboard keys you will see tasks switching.

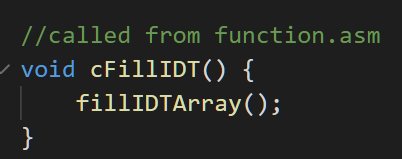


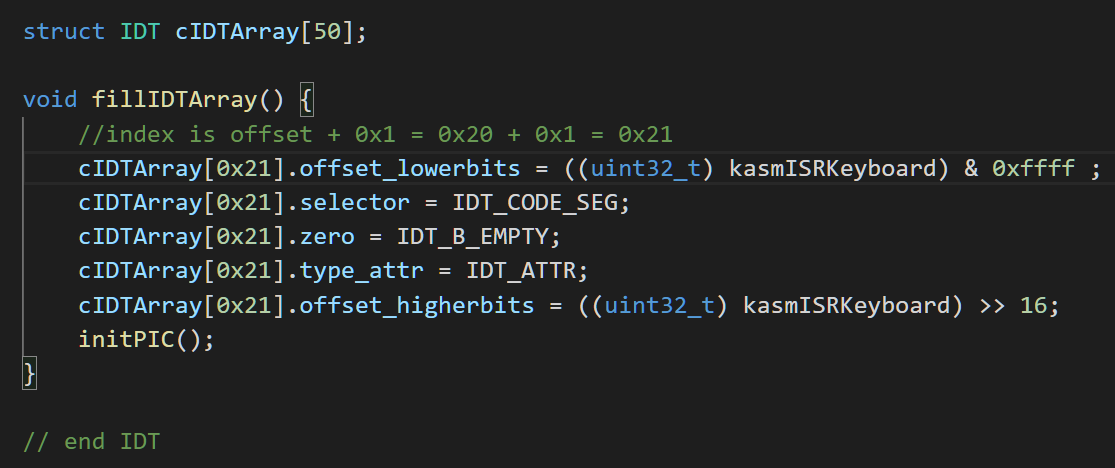
Let’s walk through this…the first thing we need is to setup our IDT for the trap routines.

“There is one important detail left out of this discussion: how does the trap know which code to run inside the OS? Clearly, the calling process can’t specify an address to jump to (as you would when making a procedure call); doing so would allow programs to jump anywhere into the kernel which clearly is a Very Bad Idea. Thus, the kernel must carefully control what code executes upon a trap. **The kernel does so by setting up a trap table at boot time**. When the machine boots up, it does so in privileged (kernel) mode, and thus is free to configure machine hardware as need be. One of the first things the OS thus does is to tell the hardware what code to run when certain exceptional events occur. For example, what code should run when a hard-disk interrupt takes place, **when a keyboard interrupt occurs**, or when a program makes a system call? The OS informs the hardware of the locations of these trap handlers, usually with some kind of special instruction**. Once the hardware is informed, it remembers the location of these handlers until the machine is next rebooted**, and thus the hardware knows what to do (i.e., what code to jump to) when system calls and other exceptional events take place.” – **pg 4,5.**

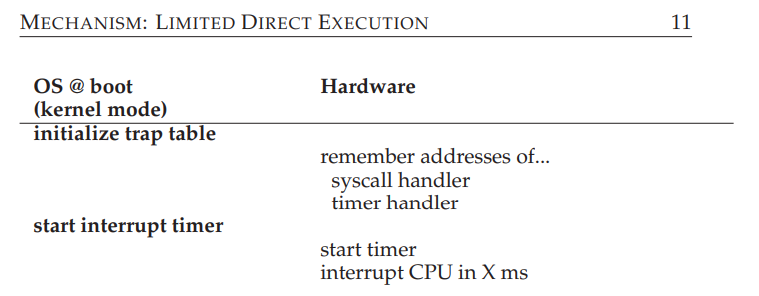




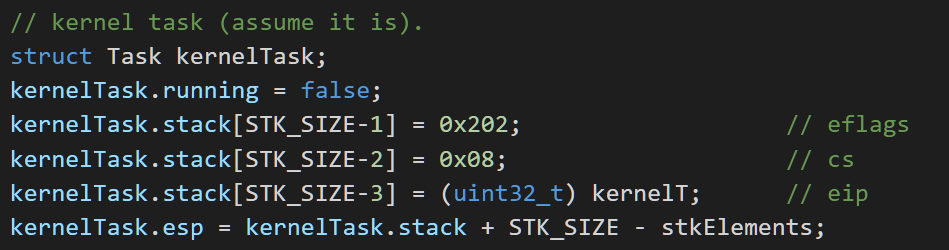




Now we are done with IDT setup on OS bootup, except instead of timer interrupt we are using keyboard interrupt.



Initially on the kernel is running which is why you see the first red screen above. Note: all processes in this example are running in kernel mode.





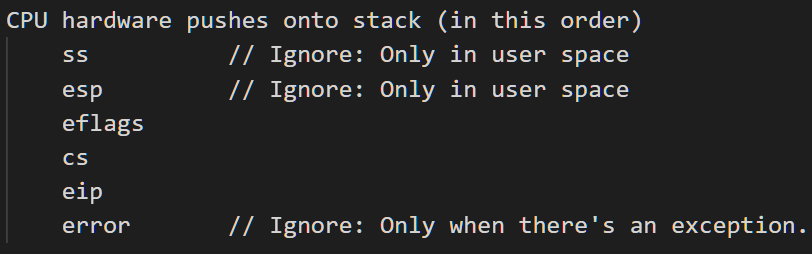
Question – Which stack are we using *at this point*? We are using the actual **kernel OS stack** setup at boot time.



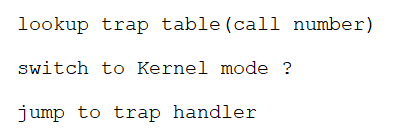
Now, you press a key, and INT occurs and instantly we break out of the kernel loop



Also at this instance, the HARDWARE (QEMU or BOCHS) will push the following onto the OS kernel stack, look for the trap routine (which is supposed to handle this INT) in the trap table and jump to it.

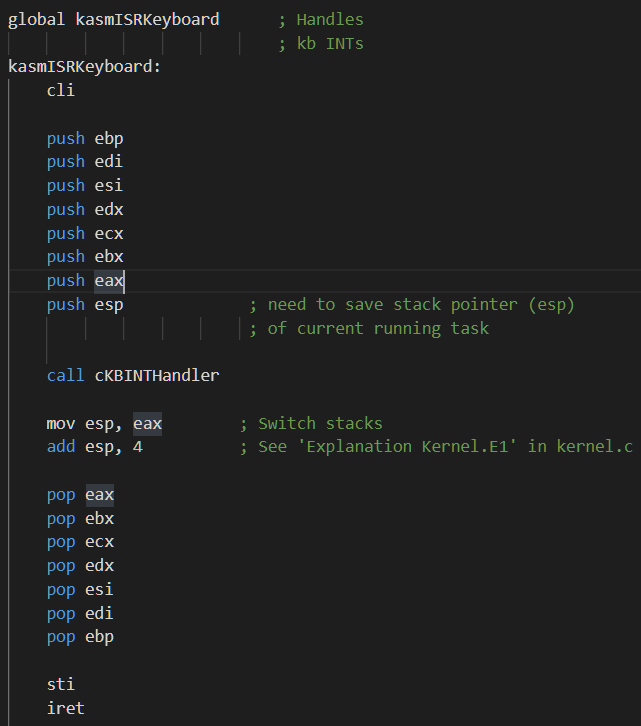
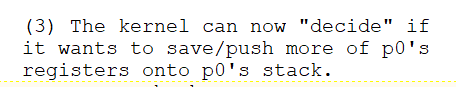


…then Lookup the trap table

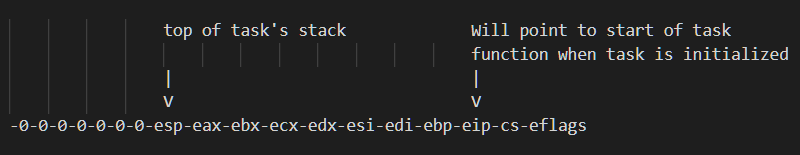


Question - Do we need to switch to kernel mode? Does qemu/bochs switch to kernel mode?? I don’t know the answer to this yet.

Trap handler (ISR) – *Your* Kernel/OS now takes over



In the first part, **just before we call cKBINTHandler in kernel.c**, 11 values are being pushed onto the *kernel* stack

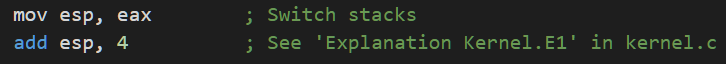
­ 

Thus, we have successfully saved the kernel’s OWN registers onto its OWN stack!

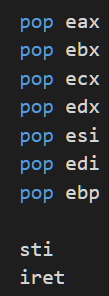
Now we call **cKBINTHandler** where we switch to the next task which is called **kernelT**. If it’s not running, start it, and then return that task’s stack (ESP).



Then, in the second part of the ISR, we switch from the OS stack, to this new kernelT’s stack.



… followed by popping the values off of kernelT’s stack and onto the kernel/OS registers, before **finally returning from trap (iret)**



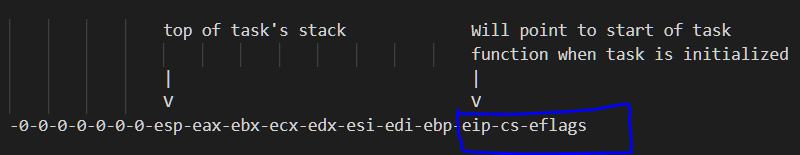
When we return from trap routine, the HARDWARE takes over, and pops the following off the *CURRENT* stack (which, as you must have guessed, is the KernelT’s private stack). EIP is now pointing to the start of kernel task. In essence, EIP is always pointing to the saved state of the running process *if it was running*, or its start location if *it was not running*.

pop eip (the very important instruction pointer).

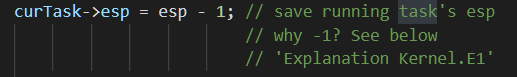
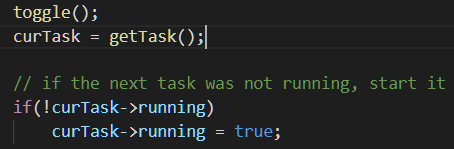
pop cs

pop eflags

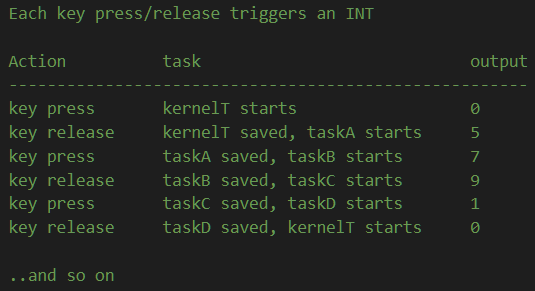
The hardware will now jump to whatever EIP is pointing to 😊



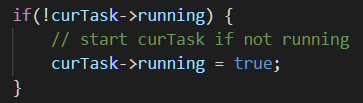
Now, KernelT is running, and a few seconds later, you press another key (or release the previously pressed key). The same steps are repeated.

* eflags, cs, and eip are pushed (saved) onto KernelT’s stack by the hardware
* hardware looks up the trap routine for this INT and jumps to it
* ebp, edi, esi, edx, ecx, ebx, eax, esp are pushed (saved) onto KernelT’s stack by the OS (kernel)
* a call is made to **cKBINTHandler** 
  + **cKBINTHandler** gets the current running task (kernelT) and determines that it is already running.
  + kernelT’s esp gets the esp that was passed to **cKBINTHandler** which was the LAST item pushed onto the stack just before call to **cKBINTHandler** (see above), *minus 1*.
  + 
  + We now get the next task and start it if it was not running (this will be **taskA**)
  + 
  + Finally, we return taskA’s esp
  + 
* Now we have returned from **cKBINTHandler** and must switch from kernelT’s stack to taskA’s stack
* 
* The OS then replaces the kernelT’s registers with taskA’s registers by popping off:
* eax, ebx, ecx, edx, esi, edi, ebp
* and finally **return from trap**
* 
* The hardware is back in charge now and pops off the remaining values off of taskA’s stack
* eip, cs, eflags
* Hardware now jumps to whatever EIP is pointing to 😊
* And so now taskA is running!

Above is repeated each time a keyboard INT occurs and we keep switching to the next task as shown here:



You will have observed that the following if condition is only executed once (on the FIRST keypress i.e. keyboard interrupt). Each subsequence keypress/release will fall in the else part of the **cKBINTHandler** function.



Another observation is that the original kernel.c *while loop* (the original kernel code) never runs again after that first keyboard interrupt. Why? The answer should be obvious 😊

