CJ Bernstein, Julia Edholm, and Emily Masten

COSC 261

Spring 2015

Project 2

Professor Kaplan:

If you run our bios, kernel, and init with user-program-1, user-program-2, and user-program-3, you should expect to see user-program-1 complete and exit, user-program-2 complete and exit, and user-program-3 throw a DIVIDE\_BY\_ZERO interrupt and exit. These three processes will run in steps of size 10 as a result of the functioning clock alarm. We have set up the code to print to the console when anything of significance happens so if you “showconsole text” after running several thousand steps (try 10,000), you should see a play by play of everything that happened, including a message indicating an error free shut down. Below are some descriptions of how we went about certain tricky components of the kernel and then a guide to all of our functionalities and how many steps to take to get to an example of that functionality in our code.

**Interrupt handlers:** Most of the interrupt handlers simply print the interrupt, exit the process, and schedule a new process. There are two exceptions.

*SYSTEM\_CALL:* We have implemented EXIT (1), CREATE (2), GET\_ROM\_COUNT (3), and PRINT (4). When the process makes a SYSC, it must store the system call code referenced above in %G0. Any additional information (ie the ROM number of the process to CREATE) is stored in %G1. Any information returned as a result of the call (ie the number of ROMs for GET\_ROM\_COUNT) is returned in %G0. In the case of a SYSC, we add 16 to the IP before jumping back into the kernel.

*CLOCK\_ALARM:* CLOCK\_ALARM pauses the current process (by storing all necessary information in the process table) and then schedules a new process. We SETALM every time we enter or exit the kernel. When entering the kernel (or init) we set srcA on SETALM to our two word value that contains -1 (offset\_kernel) so that there will not be any CLOCK\_ALARM interrupts thrown while the kernel is running. When entering user processes (exiting the kernel), we set srcA to the main memory address for our two word value (process\_offset) that contains 10 right before we JUMPMD. We set srcB to 2 (a non-zero value) at all times because we want the value in srcA to be an offset. In the console, you will see text indicating every time a CLOCK\_ALARM interrupt is thrown and that the code continues as normal to indicate that the pausing and re-scheduling of processes functions correctly.

**Kernel level interrupts:** A kernel level interrupt will trigger a MEGA\_HALT. The console will show a message that there was a kernel level interrupt and the system will HALT. We do this by using a static kernel\_indicator that is turned on (1) while we are running kernel code and turned off (0) while we are running user level code (including init because init throws interrupts that we need to work i.e CREATE, EXIT, GET\_ROM\_COUNT). Every time an interrupt is thrown, the first thing the kernel does is check to see if the kernel\_indicator is 1. If so, MEGA\_HALT. Otherwise, it turns the kernel\_indicator to 1 and continues. Just before the JUMPMD into a process, we switch the kernel\_indicator to 0. The kernel\_indicator starts at 1 since we start in kernel code.

**Process table:** Since we don’t have a heap, we created a process table of size eight in the statics. Note that you will not be able to run more than init plus seven ROMs on this system without increasing the size of the process table. The end of the process table is signaled by a static that holds the value 27. If the process table is ever full, it will throw a MEGA\_HALT. If the process table is empty, you will see a message in the console that says “Finished running all processes” to indicate an error free shut-down and the system will halt.

**Important functionalities:** *\*\*note that we do a lot of printing to the console which is why this takes so many steps (it takes about 40 steps to print each character of a string)* **ALL STEPS ARE FROM THE INTIAL RUN COMMAND**

* Loading init [Step ~695]
* System calls
  + Using GET\_ROM\_COUNT [Step ~1113 to see an example from init]
  + Using CREATE [Step ~1253 to see an example from init]
  + Using EXIT [Step ~3881 to see an example from init]
  + Using PRINT [Step ~699 to see an example from init]
* JUMPMD into a process with virtual addressing [Step ~4695 is where we JUMPMD to user-program-1]
  + Every time we JUMPMD we do the following:
    - SETBS, SETLM, and SETALM
    - Update current\_process\_ID (this should remain the same as when the interrupt was thrown except for in the case of a CLOCK\_ALARM or EXIT in which case a new process is scheduled in \_schedule\_new\_process)
    - Change kernel\_indicator from 1 to 0 to indicator the change to a user level program
    - Turn on virtual addressing and change permissions with JUMPMD [destination virtual address, NOT main memory address] 6
      * The 6 is 0110 so halt flag is clear, user/supervisor mode flag is set to user mode, virtual/physical addressing mode flag is set to virtual, and paged addressing mode flag is clear
* Interrupts [Step ~7987 to see an example of a DIVIDE\_BY\_ZERO from user-program-3]
* CLOCK\_ALARM [After starting user-program-1, if you step in chunks of ~2000 you will see CLOCK\_ALARMs happening all the time]