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 rom1 complete, rom2 throw a DIVIDE\_BY\_ZERO interrupt and exit, and rom3 complete. These three processes will run in steps of size xx as a result of the clock alarm. We have set the code up 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 that the code ended in a halt and….

**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. In the case of any other interrupt, we do not add 16 to the IP because we need to redo that instruction (hypothetically, here of course we are just exiting the process since we cannot fix the problems that are causing the interrupts).

*CLOCK\_ALARM:* CLOCK\_ALARM pauses the current process (by storing all necessary information in the process table) and then schedules a new process. [How do we handle GETCLK and SETALM]

**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:**

* Loading init
* System calls
  + Using GET\_ROM\_COUNT from init
  + Using CREATE from init
  + Using EXIT from init
  + Using PRINT from init
* JUMPMD into a process with virtual addressing
  + SETBS and SETLM
  + 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
* CLOCK\_ALARM