CSC 262
Homework #5

Due at the start of class on Wednesday, March 14.

1. **Context Switching**. As discussed in class, when a process running on a Pentium chip is interrupted, the Pentium stores the current state within the process table entry for the current process. Older chips used a more simple strategy: on an interrupt, all registers that form part of the process state are dumped by the hardware into a specific area of memory reserved for the purpose (say, memory addresses 0010 to 001F). Among other things, this saves some space in the process table and is slightly easier to implement. This question is designed to make you think about why the Pentium designers may have chosen to use a more complex design.

a. The Pentium allows different priorities to be assigned to different types of interrupts. High-priority interrupts are allowed during the handling of a lower-priority interrupt. Explain why this would be a problem with the simpler chip design described above. Can you design a way to work around the shortcoming? (Assume that interrupts are disabled when an interrupt occurs, but we would like to enable them again as soon as possible. Similarly, a return from interrupt instruction reenables interrupts if necessary.) Hint: simulate the entire interrupt sequence by hand, keeping track of what information is stored where.

b. Multiple processor chips may sometimes be combined to create a parallel computer. Two configurations are common: each processor may have its own bus and private memory, or all the processors may share a common bus and memory. In any case, suppose that each processor can disable interrupts only for itself, but not for the other processors.[1] Which memory configuration would you recommend if the processor chips use the simpler design described above, and why? Would the workaround you proposed above work in this case also?

2. **Process Scheduling**. The diagram below shows relevant parts of a system's process table at a particular point in time. The **Next** entry is used to store two circular threaded linked lists, one for ready/running processes and one for blocked processes.

{Stack Area}	*{Stack Area}*	*{Stack Area}*	*{Stack Area}*	*{Stack Area}*
Process ID: 0	Process ID: 1	Process ID: 2	Process ID: 3	Process ID: 4
Status: Ready	Status: Running	Status: Ready	Status: Blocked	Status: Ready
Priority: 20	Priority: 40	Priority: 15	Priority: 50	Priority: 25
Next: 2	Next: 4	Next: 1	Next: 3	Next: 0

a). Suppose that round robin scheduling is used, and a timer interrupt occurs. What process would run next?

b). Suppose instead that process 1 is interrupted because it requests a resource that is busy. Draw the updated process table after a new process has been started, again assuming round robin scheduling.

c). Suppose that priority scheduling is used instead. What process would be selected to run after a timer interrupt? After a request for a busy resource?

[1] Mechanisms for disabling interrupts across chips have been devised, but they are expensive to implement.